University of Zurich[UZH]

Fine-Grained Code Changes and Bugs

Improving Bug Prediction

A dissertation submitted to the Faculty of Economics,
Business Administration and Information Technology
of the University of Zurich

for the degree of
Doctor of Science

by
Emanuel Giger
from Zurich, ZH, Switzerland

Accepted on the recommendation of
Prof. Dr. Harald C. Gall, University of Zurich, Switzerland
Prof. Dr. Andreas Zeller, Saarland University, Germany

2012

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, March 2012

Head of the Ph.D. committee for informatics: Prof. Abraham Bernstein, Ph.D.

Acknowledgements

During my PhD studies I had the opportunity to meet many great people who all contributed substantially to my work.

First of all, I would like to express my heartiest gratitude to my advisor Harald C. Gall for giving me the unique opportunity to pursue my PhD studies at the University of Zurich. I'm very grateful for his permanent support and kindness—from the beginning of my work until the very end— and in particular for creating such an outstanding team spirit. I was able to profit greatly from his support, knowledge, and experience.

Special thanks go to Andreas Zeller for accepting to be part of my committee as an external examiner and for coming all the way to Zurich for my defense. Already during my PhD studies I had the pleasure to attend and enjoy some of his exceptional talks, which greatly influenced my research.

Many thanks go to Martin Pinzger who contributed (statistically ;)) significantly to this dissertation. Even after moving to Delft he still found time to discuss many research opportunities that resulted in publications that build the core foundation of my work. I also greatly enjoyed our surf session at the shores of Hawai'i.

Thanks go to Marco D'Ambros for sharing with me his extremely valuable and beneficial knowledge about bug prediction, and especially for taking us to some delicious restaurants in Lugano during the SVC project sessions.

Many many thanks go the my colleagues from the SEAL lab at the University of Zurich. I truly appreciated their very helpful comments, suggestions, and discussions. They always encouraged me to go on with this long "PhD-Quest". It was an honor to be a member of such a heroic raid. The time was truly epic, and I could not think of any better group.

Abstract

Software development and, in particular, software maintenance are time consuming and require detailed knowledge of the structure and the past development activities of a software system. Limited resources and time constraints make the situation even more difficult. Therefore, a significant amount of research effort has been dedicated to learning software prediction models that allow project members to allocate and spend the limited resources efficiently on the (most) critical parts of their software system. Prominent examples are bug prediction models and change prediction models: Bug prediction models identify the bug-prone modules of a software system that should be tested with care; change prediction models identify modules that change frequently and in combination with other modules, *i.e.*, they are change coupled. By combining statistical methods, data mining approaches, and machine learning techniques software prediction models provide a structured and analytical basis to make decisions.

Researchers proposed a wide range of approaches to build effective prediction models that take into account multiple aspects of the software development process. They achieved especially good prediction performance, guiding developers towards those parts of their system where a large share of bugs can be expected. For that, they rely on change data provided by version control systems (VCS). However, due to the fact that current VCS track code changes only on file-level and textual basis most of those approaches suffer from coarse-grained and rather generic change information. More fine-grained change information, for instance, at the level of source code statements, and the type of changes, *e.g.*, whether a method was renamed or a condition expression was changed, are often not taken into account. Therefore, investigating the development process and the evolution of software at a fine-grained change level has recently experienced an increasing attention in research.

The key contribution of this thesis is to improve software prediction models

by using fine-grained source code changes. Those changes are based on the abstract syntax tree structure of source code and allow us to track code changes at the fine-grained level of individual statements. We show with a series of empirical studies using the change history of open-source projects how prediction models can benefit in terms of prediction performance and prediction granularity from the more detailed change information.

First, we compare fine-grained source code changes and code churn, *i.e.*, lines modified, for bug prediction. The results with data from the Eclipse platform show that fine grained-source code changes significantly outperform code churn when classifying source files into bug- and not bug-prone, as well as when predicting the number of bugs in source files. Moreover, these results give more insights about the relation of individual types of code changes, *e.g.*, method declaration changes and bugs. For instance, in our dataset method declaration changes exhibit a stronger correlation with the number of bugs than class declaration changes.

Second, we leverage fine-grained source code changes to predict bugs at method-level. This is beneficial as files can grow arbitrarily large. Hence, if bugs are predicted at the level of files a developer needs to manually inspect all methods of a file one by one until a particular bug is located.

Third, we build models using source code properties, *e.g.*, complexity, to predict whether a source file will be affected by a certain type of code change. Predicting the type of changes is of practical interest, for instance, in the context of software testing as different change types require different levels of testing: While for small statement changes local unit-tests are mostly sufficient, API changes, *e.g.*, method declaration changes, might require system-wide integration-tests which are more expensive. Hence, knowing (in advance) which types of changes will most likely occur in a source file can help to better plan and develop tests, and, in case of limited resources, prioritize among different types of testing.

Finally, to assist developers in bug triaging we compute prediction models based on the attributes of a bug report that can be used to estimate whether a bug will be fixed fast or whether it will take more time for resolution.

The results and findings of this thesis give evidence that fine-grained source code changes can improve software prediction models to provide more accurate results.

Zusammenfassung

Die Entwicklung und insbesondere die Wartung heutiger Software Systeme sind zeitintensiv und erfordern in hohem Masse detaillierte Kenntnisse über deren innere Struktur sowie über die historischen Änderungen und Designentscheide solcher Systeme. Dieser Umstand wird zusätzlich erschwert durch strenge Zeitvorgaben und limitierte Entwicklungsressourcen. Die Forschung im Bereich der Softwareentwicklung konzentriert sich daher vermehrt auf die Erforschung von Methoden, die einen möglichst effizienten Einsatz der verfügbaren Ressourcen erlauben und die Wartung bestehender Software Systeme als Ganzes vereinfachen und beschleunigen. Bekannte Beispiele für solche Methoden sind Fehler- und Änderungsvorhersagemodelle: Fehlervorhersagemodelle sollen mit möglichst hoher Zuverlässigkeit die fehlerhaften Module einer Software (im Voraus) identifizieren, damit diese eingehend getestet werden können. So können zum Beispiel Testkapazitäten von fehlerfreien Modulen hin zu besonders fehleranfälligen Modulen verlagert werden; Änderungsvorhersagemodelle sollen möglichst genau die Module bestimmen, die mit hoher Frequenz geändert werden müssen. Da Änderungen am Programmcode immer mit Kosten und potentiell mit Fehler verbunden sind, stellen solche Module Kandidaten für Gegenmassnahmen dar, um in Zukunft die Anzahl Änderungen tief zu halten.

In der Literatur existieren zahlreiche Ansätze für die Erzeugung solcher Vorhersagemodelle. Die meisten dieser Ansätze verwenden hierfür historische Daten der Entwicklungsgeschichte eines Systems. Diese Daten werden von speziellen Programmen für Versionierung von Programmcode, z.B. CVS, SVN oder Git, zur Verfügung gestellt. Heutige Versionierungsprogramme verwalten Änderungen am Programmcode jedoch nur auf der Ebene von einzelnen Dateien. Des Weiteren wird Programmcode als Text behandelt, und dessen implizite Struktur wird daher weitgehend ignoriert. Bestehende Vorhersagemodelle basieren daher auf relativ groben und generischen Änderungsdaten.

Änderungen auf fein granularer Ebene, z.B. auf der Ebene einzelner Instruktionen, sowie der Typ der Änderungen, z.B. wurde die Bedingung einer Schleife angepasst oder der Name einer Methode geändert, werden daher in der Regel nicht berücksichtig für solche Modelle. In jüngster Zeit wurde dieses Defizit erkannt, und die Forschung hat sich vermehrt auf die Analyse von Änderungen am Programmcode auf fein granularer Ebene konzentriert.

Ziel der vorliegenden Arbeit ist es, Vorhersagemodelle im Bereich der Softwareentwicklung durch den Einbezug von feinkörniger und strukturbedingter Änderungsinformation zu verbessern. Wir zeigen in einer Reihe von empirischen Studien die Vorteile solcher Modelle an Hand der Entwicklungsgeschichte bekannter Open-Source Systeme.

Im ersten Schritt vergleichen wir dazu die Genauigkeit von Vorhersagemodellen basierend auf feinkörniger Änderungsdaten und von solchen, die auf herkömmlichen Änderungsdaten basieren. Die Resultate zeigen, dass unsere Modelle die fehlerhaften Module eines Systems statisch signifikant besser identifizieren können. Zusätzlich können explizit die empirischen Zusammenhänge einzelner Änderungstypen und der Fehleranfälligkeit eines Moduls aufgezeigt werden.

In einem zweiten Schritt benutzen wir fein granularen Änderungsdaten, um Fehler auf der Ebene von Methoden anstatt auf der Ebene von Dateien zu identifizieren. Dies hat den Vorteil, dass anstatt ganzer Dateien nur gewisse Methoden inspiziert werden müssen, um einen Fehler zu lokalisieren und zu beheben. Dies insbesondere dann von Nutzen, wenn Dateien, die fehlerhaften Programmcode enthalten, sehr gross werden, da der Suchraum erheblich eingeschränkt wird.

Als dritten Schritt bilden wir Modelle, welche es uns erlauben die Typen von Änderungen vorherzusagen. Dies hat zum Beispiel für die Manager eines Software Projektes den Vorteil, dass sie ihre Testvorgänge besser planen können: Sind in erster Linie Änderungen an der Programmschnittstelle zu erwarten, können rechtzeitig genügend Ressourcen für aufwendige Schnittstellen- und Integrationstests mit anderen, abhängigen Modulen eingeplant werden. Kommen jedoch vor allem kleine Änderungen vor, sind in der Regel einfache Unit-Tests ausreichend.

Zum Abschluss entwickeln wir Modelle, die den Entwickler beim Beheben von Fehlern in seiner Software unterstützen sollen. Diese Modelle erlauben eine Einschätzung, ob ein bestimmter Fehler länger benötigen wird zur Behebung oder in relativ wenig Zeit behoben werden kann.

Die Resultate dieser Arbeit zeigen auf, dass feinkörnige Änderungen sowie die Information über den Typ bestimmter Änderungen Vorhersagemodelle für

die Entwicklung von Software Systemen signifikant verbessern können.

Contents

List of Figures

List of Tables

1

Synopsis

1.1 Introduction

SOFTWARE systems are designed to support and automate (business) processes. Those processes are constantly adapted to meet changes in the business environment, *e.g.*, new legal regulations, extensions of the current product line, or opening up new market segments. To remain useful to their stakeholders in such a dynamic context software systems have to be continuously adapted alongside to the underlying business processes. Furthermore, the (initial) requirements of a software system are never complete once it is released; new requirements and features are demanded over time, and they have to be integrated in running and productive systems—inducing a revisiting, iterative approach [Bro95] (see Figure 1.1). In reality, each stage in Figure 1.1 can be further divided in multiple non-sequential substeps as illustrated, for instance, by *Software Maintenance*. Moreover, often certain details and properties of a system become clear and concrete during the implementation itself making an adaption of the initial design of the software necessary [PC86].

As a matter of fact, software systems are often long-living legacy systems

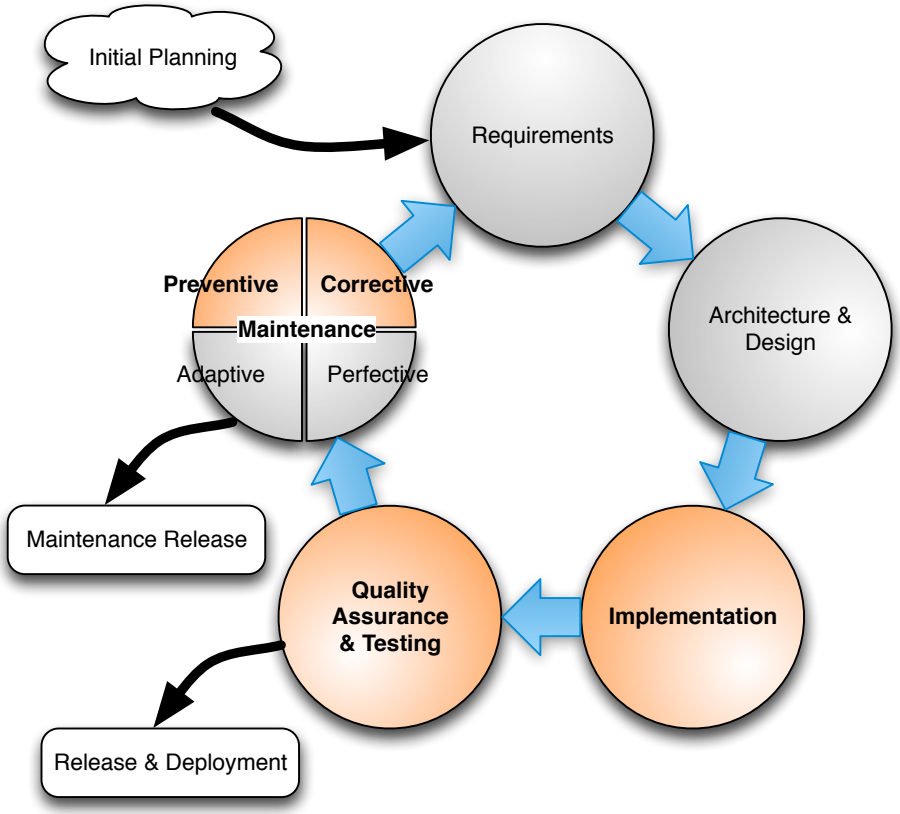


Figure 1.1: Schematic Overview of the Iterative Software Development Stages. The colored circles indicate the stages to which the results of this thesis contribute.

that constitute an essential and critical business asset to their stakeholders through decades. Therefore, it is absolutely necessary that those software systems are supported and maintained in a disciplined, structured manner that allows for their continued and error free use.

Given this fundamental importance, the maintenance of legacy systems is an integral part of software development as it ensures the sustainability of those systems during their life-time [Pig96]. Moreover, Sommerville estimates that there are around 250 billion lines of code being maintained [Som00]. This number stresses the key role of software maintenance.

Software maintenance, however, is expensive and a major cost driver for

software development. For instance, two third of the total software development effort is spent on existing software systems rather than implementing new ones [McK84]. Moreover, as quantified in [RN78] the costs spend on maintenance account for one third of the total development costs; other studies state that those costs exceed 50% of the total costs and are continuously increasing, *e.g.*, [Boe76, PA06]. Consequently, more recent studies claim that maintenance constitutes for up to 90% of the entire software life-cycle costs, *e.g.*, [Erl00].

To better cope with software maintenance and, hence, to reduce its costs research proposed a wide range of approaches that support developers. Examples are, methods to quickly understanding and visualizing the architecture of a legacy software, models to bridging the gap between the design and the implementation of a software system, methods for identifying the *hot spots* in a system, tools that support automated refactoring and track the corresponding changes, approaches to handle change coupled modules, tools to facilitate regression testing, and empirical models that make use of data analytics to provide decision support. Some research venues, such as the *International Conference on Software Maintenance*[1], the *European Conference on Software Maintenance and Reengineering*[2], and the *Working Conference on Reverse Engineering*[3], focus especially on software maintenance and evolution.

An important part of software maintenance is *corrective maintenance* (see Figure 1.1), *i.e.*, maintenance related to fixing and correcting software bugs [Swa76]. According to a study presented in [Tas02] software errors cost the U.S. industry up to 60 billion dollars a year. This problem is even worse as the relative costs of software bugs are of a magnitude higher if bugs are discovered and fixed at a later stage in the development process, *e.g.*, after the software is released [Boe81, Tas02]. Therefore, it is crucial to identify *bug-prone* modules as early as possible. For that, Quality assurance (QA) mechanisms are required [KFN99]. One part of QA is to establish disciplined testing procedures

[1] http://selab.fbk.eu/icsm2012/
[2] http://csmr.eu/
[3] http://distat.unimol.it/wcre2012/index.html

that rigorously test all components of a software system. The goal is to fix as many bugs as possible *before* the software is released and distributed to customers, *i.e.*, to reduce the defect rate.

Like software maintenance, QA is an expensive and resource costly task since team members need to spend a significant amount of their time to inspect the entire software in detail rather than, for example, implementing new features. Additionally, if bugs are detected the fixing of those consumes further development time.

However, time pressure is a constant issue in software engineering; there is a strong demand for short release cycles and fast deployed patches providing bug fixes. This opposes certain restrictions to QA that intensify the *bug-proneness* of a system. In particular, today's software systems are large and complex, and, hence, are difficult to change. This level of increasing complexity requires that a developer, in order to make changes, possesses detailed knowledge about the internal structure of her software. Furthermore, she needs to be aware of all prior design and implementation decisions before any changes to the system are carried out. However, limited time and upcoming deadlines inhibit a developer to spend the full amount of time necessary to acquire that knowledge. As a consequence, a developer might not completely understand the software she is supposed to change. This particular lack of knowledge can lead to *ignorant surgery* and deteriorates the maintainability of a system, *i.e.*, future changes are more difficult, and therefore, are more likely to introduce new bugs [Par94]. Even changes that are intended to fix existing bugs are sometimes *bug-prone* themselves. According to [PP05], 40% of all changes to correct bugs introduced more bugs. In such a case any new bug in turn needs to be fixed causing additional workload [CH96]. This particular problem is intensified, for example, as software ages and repeatedly undergoes *ignorant surgery*; or if novice developers that have even fewer knowledge of the system are assigned with change tasks. Eick *et al.* refer to this phenomena as *Code Decay* [EGK+01]: Source code becomes more difficult to change during its life-time because previous changes decreased the maintainability.

The decreased maintainability of a software system, its increasing size

and complexity, as well as time pressure in addition with limited resources prevents development teams from rigorously testing *all* parts of their software. In other words, if there are not enough resources available to testing the entire system the resources need to be allocated most efficiently, *i.e.*, to finding as many bugs as possible. Therefore, the available resources must be focused on those components where most of the bugs are expected. This leads to the question: How to identify those parts of my software that are affected by most of the bugs?

To address this problem research has developed and proposed *bug prediction models*. Bug prediction models combine statistical methods, machine learning techniques, and business data analytics to support an efficient handling of bugs. Having access to such a structured and analytical approach that supports allocating QA resources with maximum efficiency can make an advantageous difference for a development team, in particular, when considering the fact that at present a large portion of all decisions made in companies are still based on a manager's personal perspective and "guts" [Vuj08].

Bug prediction models are a major focus of this thesis, and therefore, will be discussed in depth in Section 1.2.

1.2 Bug Prediction

The purpose of bug prediction models is to support development teams in identifying the *bug-prone* parts of a system. Those are the critical parts of a system where resources are needed the most and should be tested with care. Building accurate bug prediction models is considered as one of the holy grails in software engineering and has received much attention in research [DLR11]. A variety of approaches were proposed that leverage a multitude of statistical models and machine learning algorithms, each based on its own set of input data for training.

Despite the diversity of existing prediction models in literature they can be described by the type of their input data: *historical change metrics*, *source code*

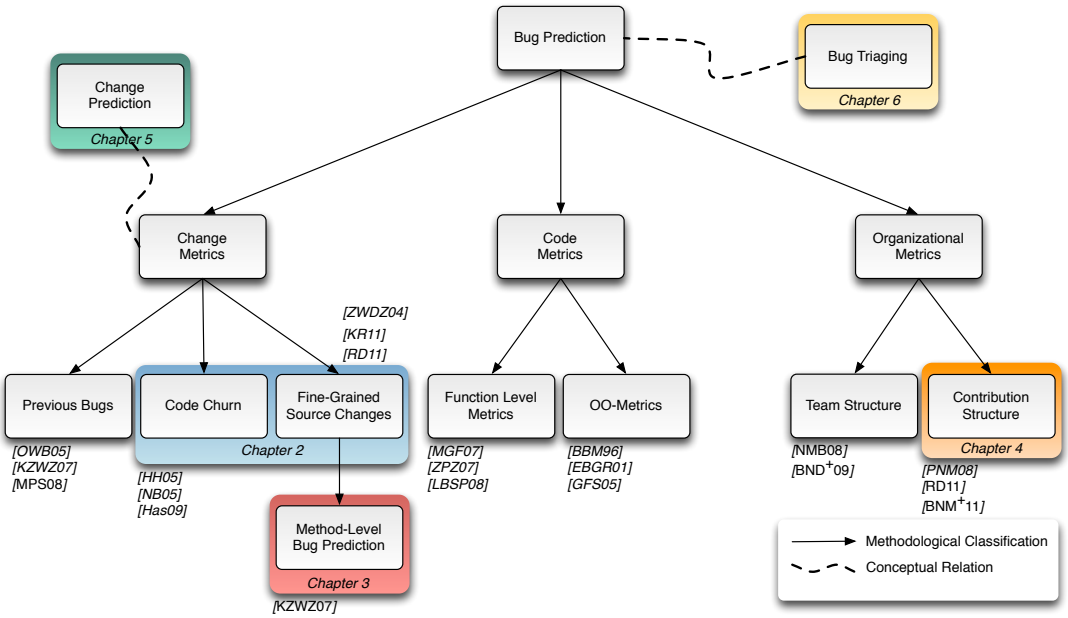


Figure 1.2: State-of-the-art overview of bug prediction models as classified by their input data, selected related work, and the thesis' chapters.

metrics, or *organizational metrics*—or any combination of these metric families.

Figure 1.2 gives an overview of the basic bug prediction approaches as classified by their input data, selected related work, and the chapters of this thesis.

1.2.1 Change Metrics

Change metrics (also referred to as *process* or *code churn metrics*) represent the development history of a software system. The rationale for using past changes as indicators for bugs (and to build bug prediction models) is that bugs are introduced by recent changes [KZPW06], *e.g.*, due to *ignorant surgery* (see Section 1.1). Thus, the more changes are done to a particular part of the source code the more likely it will contain bugs. The change history of a software is usually stored and managed in version control systems (VCS), *e.g.*, CVS, SVN, GIT, or Mercurial. Prominent change measures used for prediction

model training are *file revisions* and *lines added/deleted/changed* that are obtained by applying a *text-diff* algorithm on two subsequent versions of a source code file.

The emergence of open-source projects in the past decade made a large amount of such data publicly accessible to a wide audience. Moreover, bug-tracking systems, *e.g.*, Bugzilla, mailing lists, and online discussion boards provide additional information regarding the development process of a particular project. The increased availability of those data sources opened up the opportunity for a new research field called *Mining Software Repositories* (MSR)[4] (see [KCM07] for a survey of current MSR approaches). The general purpose of MSR is to leverage the data extracted from software repositories to analyze the software engineering process, support software maintenance, and understand how software evolves over time. Bug prediction is only one discipline (but probably the most active) of MSR; others are, for example, studying the structure of open-source communities, understanding the communication between project members, automated generation of test-cases, empirical studies, data visualization, or tool support.

Among the first to use past changes for bug prediction model building were Khoshgoftaar *et al.* [KAG+96]. By using the change history of a legacy software system for telecommunications the authors showed that a high code churn, *i.e.*, a high amount of lines added and deleted, is a good indicator for *bug-prone* modules. In [GKMS00], Generalized Linear Models were built based on several change metrics, *e.g.*, number of changes or average age of the code. The best model used the sum of the contributions of all changes made to a module as predictor.

Nagappan and Ball showed that relative change metrics, *e.g.*, code churn weighted by lines of code, are better indicators than absolute values when predicting the defect density of Windows Server 2003 binaries [NB05]. Hassan and Holt presented a *Top Ten List* of those directories that most likely contain bugs [HH05]. The list is dynamically adapted based on the most recent change

[4] http://2012.msrconf.org/

and bug data. Their algorithm was validated on several open-source system. Hassan quantified the complexity of source code changes using *Shannon's entropy* [Has09]. The more complex source code changes are, the more likely they are bug-prone.

Moreover, research investigated how measuring source code changes over different timeframes affects the accuracy of bug prediction models. For instance, Nagappan *et al.* found that the number of subsequent, consecutive changes (called *change bursts*) rather than the total number of changes is a strong predictor for bugs [NZZ+10]. Furthermore, Ekanayake *et al.* showed with a dataset of the change history of several Eclipse plugins that temporal features, such as the *number of file revisions measured within a 1 month timeframe before the release*, improves prediction performance [ETGB].

Typically, the goal of bug prediction models is to identify the bug-prone modules of a software system, *i.e.*, the location of bugs. Whether a module is labeled as *bug-prone* is determined by a pre-defined threshold, *e.g.*,"at least one bug" [MGF07], or the lower bound [NB05] and the median [GPG11] of the distribution of bugs.

A different goal is to predict the actual number of bugs in a software system. In its simplest form this is achieved by linear regression models. However, studies showed that due to the nature and distribution of software data linear regression models are not optimal , *e.g.*, [BEP07]. Hence, several non-linear models based on the change and fault history of a system were proposed. Examples are, generalized linear models (GLM) [GKMS00], regression tree models [BEP07], asymptotic regression models [GPG11], and negative binomial regression [OWB05].

1.2.2 Code Metrics

Code metrics (also referred to as *product metrics*) are directly computed on the source code itself. Hence, in contrast to change metrics (see Section 1.2.1) code metrics do not require access to the change history. Using code metrics to predict bugs assumes that a more complex and larger component of a system

is more difficult to understand and to change. Therefore, such a component is likely to contain more bugs and changing such a component is more likely to introduce further bugs [DLR11]. In the literature, two traditional suites of code metrics exist.

First, the *CK metrics suite* as defined by Chidamber and Kemerer [CK94]. This suite consists of six metrics that measure the size and complexity of various aspects of object-oriented source code and are calculated at the class level. Basili *et al.* investigated the impact of the CK object-oriented metrics suite on software quality [BBM96]. El Emam *et al.* showed that class size is a confounding effect when examining the relation between the CK suite and the bug-proneness of a class [EBGR01]. In [GFS05], the metric *Coupling Between Object classes* (CBO) of the CK suite performed the best in predicting the bug-proneness of classes of the Mozilla project. In contrast, the *Number of Children* (NOC) metric cannot be used as indicator for the bug-proneness of a class. The same suite was applied to Java and C++ source code to predict bugs in a commercial e-commerce application [SK03]. The CK suite can be extended by additional (class-level) object-oriented metrics, *e.g.*, number of static methods [ZPZ07] or inheritance based coupling [BWIL99].

The second suite consists of metrics that are not limited to object-oriented source code but are calculated on method/function level, *e.g.*, lines of code (LOC) or complexity. When applied to files, these metrics are typically averaged, summed up over all methods that belong to a particular file, or the highest value in the file is selected [ZPZ07, LBSP08].

In [OA96], a set of control flow graph based metrics and McCabe's Cyclomatic Complexity were used to predict bug-prone modules. Binkley and Schach proposed a coupling dependency metric (CDM) in [BS98]. In a case study with systems written in different languages CDM outperformed several other metrics, *e.g.*, LOC, when predicting run-time failures. Naggapan *et al.* applied a set of function level metrics to predict post-release bugs in Microsoft systems [NBZ06]. A set of complexity and size metrics was used to predict post-release bugs in releases of Eclipse [ZPZ07]. The dataset of this particular study was made publicly available by the authors and served as

baseline for further studies, *e.g.*, [MPS08]. The usefulness of code metrics to build prediction models was demonstrated using the NASA dataset [MGF07]. In [LBSP08], an extensive study was conducted with the same dataset, focusing on evaluating different machine learning algorithms. The authors state the conclusion that the difference between those algorithms is mostly not (statistically) significant. The practicability, in particular, of lines of code (LOC) to predict defects was demonstrated in [Zha09].

Software metrics—both change and code metrics—are rarely used in isolation but instead are often combined for building bug prediction models. The goal is to either achieve (significantly) higher prediction results or to study which of the metrics are better predictors for bugs. For instance, Nagappan *et al.* found out that there is no single set of metrics that stably predicts bugs over several projects [NBZ06]. In [KPB06], the J48 decision tree algorithm was used with a combination of product and process measures to predict the defect density of Mozilla releases. The results showed that process measures are good predictors. Similarly, Graves *et al.* found out that the number of changes and the age of a module yield better prediction models than product measures [GKMS00]. In a comparative study of several Eclipse releases change metrics were more efficient bug predictors than code metrics [MPS08]. Moreover, Kamei *et al.* found out that change metrics achieve better prediction performance in the case of effort aware prediction models, *i.e.*, bug prediction models that take into account the effort required for testing and code reviews [KMM+10]. Denaro and Pezzè computed multivariate prediction models using 38 different code metrics. The models were trained with data from the Apache 1.3 project and validated on the Apache 2.0 project [DP02].

A critique on bug prediction approaches can be found in [FN99]. An extensive comparison and evaluation of the most common approaches to build bug prediction model is presented in [DLR11].

1.2.3 Organizational Metrics

Work on this subject investigates the organizational and social context of the software development process. Moreover, it takes into account the emerging perception that todays software projects are complex networks in which people, technologies, and artifacts show manifold interactions. The idea is to shift from a pure technical point of view to the more socio-technial characteristics of a project.

Bird *et al.* found out that sub-communities can emerge among the members of open-source (OS) projects [BPD+08]. In particular, project members have more files in common with other members from the same sub community. Social network analysis was applied to CVS information to investigate the structure and evolution of OS projects [GBR04]. Xu *et al.* studied the social network properties of the Sourceforge development community. They discovered the *small world* phenomena for software projects. In this world co-developers and active users are sustainable factors. Other studies focused on the question how to identify and characterize team members regarding their importance, expertise, and knowledge of a given project. Huang *et al.* used a *Legitimate Peripheral Participation* model to describe the interactions between developers in OS projects and divided them into core and peripheral teams [HmL05]. OS teams often consist of a small number of developers who seek knowledge beyond their own. Ohira *et al.* used collaborative filtering and social network analysis to locate expertise and knowledge across different projects [OOOM05]. Duchenaut investigated the process of newcomers becoming a core member in the Python project [Duc05]. The success of this socialization process is determined by two factors: (1) An individual learning process where newcomers acquire technical skills and project related knowledge. (2) A political process where newcomers have to gain reputation among the senior developers by demonstrating their skills and following certain established rites within the project. A study about the process of people joining OS projects and becoming active contributors was shown in [BGD+07].

Mockus *et al.* formulated several research questions that aim at under-

standing the coordination and characteristics of the development process of open-source projects [MFH02]. For instance, *"How many people wrote code for new functionality?"*. The authors investigate those questions with data collected from the software repositories of the Apache web-server and Mozilla browser projects.

Rather than understanding the (latent) social mechanisms, the purpose of using organizational measures for bug prediction is to analyze how the socio-technical context of software development affects its quality.

Pinzger *et al.* related social-network techniques to the developer contribution network of Windows Vista binaries [PNM08]. They found that if more developers contribute to a certain binary it will more likely be affected by post-release defects. Moreover, removing minor contributors from such a network affects prediction performance negatively [BNM+11]. Somewhat surprisingly, distributed development does not seem to affect software quality [BND+09]. Nagappan *et al.* defined a catalogue of metrics that quantify the organizational complexity of the development process [NMB08]. A study with data from the Windows Vista development showed that, for example, a more complex and fragmented contribution structure increases the likelihood of post-release bugs. Moreover, the results indicate that their organizational metrics are better predictors than traditional change, *e.g.*, code churn, and code metrics, *e.g.*, complexity.

1.3 The Need for Fine-Grained Change Information: Motivation and Thesis Statement

The approaches presented in Section 1.1 achieve remarkably good prediction results. Using systematic data analytics they reliably identify the *bug-prone*, and hence, the most critical parts of a software system. This provides a valuable help for development teams for allocating their resources efficiently to those

parts—especially when facing time constraints.

As previously discussed, they make use of the change data collected from Version Control Systems (VCS). However, VCS treat and manage source code as pure text and track changes on file-level, *i.e.*, file revisions. Therefore, as argued in [FWPG07] those approaches potentially suffer from two problems: *Too coarse-grained change information* and *Lack of change semantics*.

Coarse-grained change information. This problem is due to the fact that files represent the atomic change units in current VCS. On the one hand, a source code file can be arbitrarily large. On the other hand, often only small changes are committed between two file revisions. For instance, Purushothaman and Dewayne found out that 10% of all file changes involved the change of a single line of code, and 50% of all file changes involved changing $<= 10$ lines of code [PP05]. Moreover, in Eclipse a substantial amount of bugs is fixed inside a single method [FZG08]. Similarly, change data from this thesis shows that the major share of all changes concerns individual source code statements (see Table 2.3), and that around 8 different source code entities were changed per each revision. These numbers illustrate that tracking changes on file-level is rather imprecise as it does not capture all the detailed changes. Thus, there is a need to extract source code changes on a finer-grained level.

Lack of change semantics. VCS handle source files purely as text. Hence, all changes between two versions of a file are then calculated as lines added/deleted/changed. Such textual differences ignore the implicit structure of source code and do not differentiate between individual types of code changes, *i.e.*, they do not capture the semantic meaning of source code changes. For example, renaming a local variable and changing the name of a method will both likely result in "+1 line changed" but are different source code changes. Moreover, such textual changes can falsely indicate code changes although no source code entities were modified. For instance, Fluri *et al.* showed that a substantial amount of change couplings are not caused by structural source code changes [FGP05]. In [FG06], the same authors conducted a study with the change history of the Azureus project. The results revealed that a high number of lines added/deleted are due to non source code changes, *e.g.*, text inden-

tations and license header updates. Therefore, the amount of lines changed in a source file may not fairly represent the actual substance of source code changes.

To overcome these shortcomings of traditional change metrics, the idea of measuring and analyzing code changes at a fine-grained level recently received an increasing attention in research on software evolution and maintenance. For example, the ROSE tool suggests change-coupled source code entities to developers at a fine-grained level, e.g., instance variables [ZWDZ04]. Kim *et al.* used changes at the level of block-wise grouped lines of code to train bug prediction models [KZWZ07]. Robbes *et al.* used fine-grained source changes at key-stroke level to detect several kinds of distinct logical couplings between files [RPL08]. Most recently, work showed that investigating code-ownership and interactions between developers at a fine-grained level can substantially contribute to defect prediction [RD11, LNH+11]. Kawrykow and Robillard observed that a substantial amount of source code chances are *non-essential*, *i.e.*, are not related to feature modifying change tasks [KR11]. They suggest that taking into account that distinction is beneficial when analyzing change histories of software projects. For instance, prepending the keyword `this` to field access statements can cause several files to be change coupled. However, inferring such changes as part of a high-level development activity, *e.g.*, adding a new feature, might lead to inaccurate interpretations.

Fluri *et al.* proposed a tree differencing algorithm to extract fine-grained source code changes down to the level of single source code statements [FWPG07]. Their algorithm is based on the idea of comparing two different versions of the abstract syntax tree (AST) of the source code. This has several benefits: One can exactly determine which particular source entities were changed, *i.e.*, *inserted*, *deleted*, *changed*, or *moved*. Moreover, one knows the exact location and the type of every changed source code entity within the AST. For example, as depicted in Figure 1.3 it is possible to determine that (1) the condition expression `obj!=null` in the body of method `foo()` of `Class A` was updated to `obj!=null && !obj.equals(this)`, and (2) the parameter `int b` was inserted to the declaration of method `sum` from

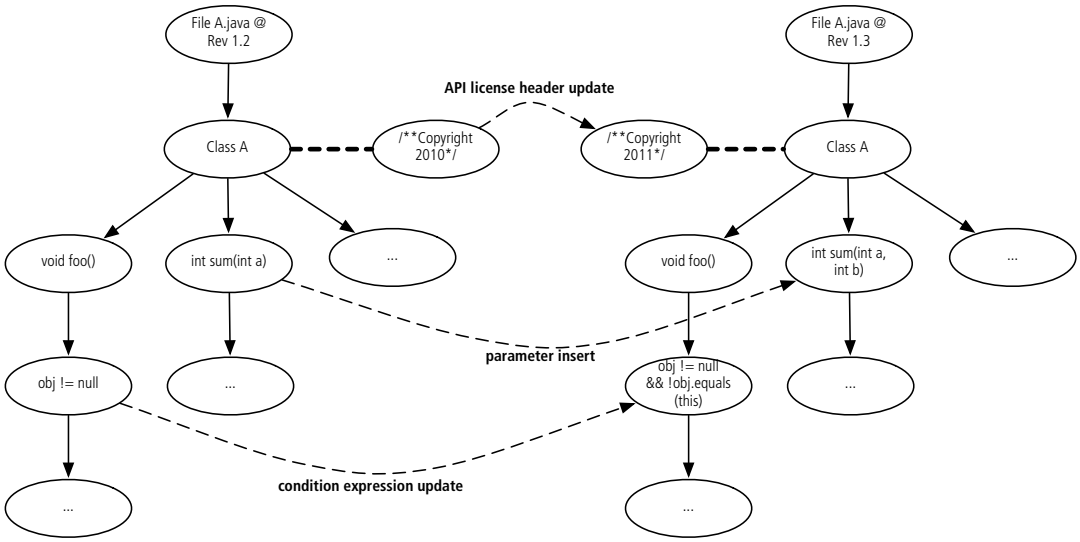


Figure 1.3: A schematic example of fine-grained change extraction based on the AST comparison of two file revisions as proposed in [FWPG07].

revision 1.2 to 1.3 of the corresponding file `A.java`. This AST based approach allows to clearly distinguish license header and API modifications from other code changes.

In this thesis we claim that measuring change data at a fine-grained level of the AST structure, *i.e.*, statement-level, and including the type semantics of source code changes is important to improving software prediction models, and hence, to provide better decision support to development teams.

To give a motivating example for how prediction models can benefit from fine-grained source code changes consider Figure 1.4. It shows the number of lines changed (Text Diff) and the number fine-grained source code chances based on AST comparison as well as the number of bugs of the Eclipse jFace plugin per half-year between 2002–2010 (numbers are normalized between [0.0:1.0]). In particular, one can see that the number of lines changed increased significantly and reached its peak during the second half of 2004 (denoted as **2004_2** in Figure 1.3). Consequently, when predicting bugs on the basis of textual code churn one would expect a similarly high number of bugs.

However, as a matter of fact, the number of bugs decreased by 0.2 from 0.4 to 0.2 compared to the first half of 2004 (denoted as **2004_1** in Figure 1.3). In contrast to code churn, the of number fine-grained source code changes exhibits a behavior likewise to the normalized number of bugs, *i.e.*, a decrease by 0.2.

A look at the revisions and commit messages of that period revealed that the high amount of lines changed is mostly caused by textual formatting and re-organization of import statements rather than by actual source code changes. Therefore, such formatting changes can potentially interfere with bug prediction models operating on textual code churn and may yield inaccurate prediction results. Fine-grained source code changes, on the other hand, can differentiate between this kind of trivial text changes and "real" source code changes.

Similarly, during the years 2006 and 2007 both bugs and fine-grained source code changes stayed at a high level while code churn remained fairly low. Again, when relating bugs only to textual changes in source code files we then might wrongly predicted a low number of bugs in that particular period, too.

In general, the plot in Figure 1.4 shows that the line representing the normalized numbers of fine-grained source code changes runs much closer to the number of bugs over the development history, especially since 2006. Therefore, we are convinced that structural, *i.e.*, AST based, fine-grained source code changes are the better indicator of bugs.

Moreover, the knowledge regarding the semantics of code changes enables an explicit quantification of the empirical relation between a particular type of change, *e.g.*, declaration changes, and the bug-proneness of a file. This allows for systematic refactorings to prevent specifically such declaration changes in the future rather than (textual) changes in general.

By building prediction models using the semantics of changes managers can further prioritize different types of testing, *e.g.*, integration tests for declaration changes, branch testing when changing conditional expressions, or localized unit tests for statement changes.

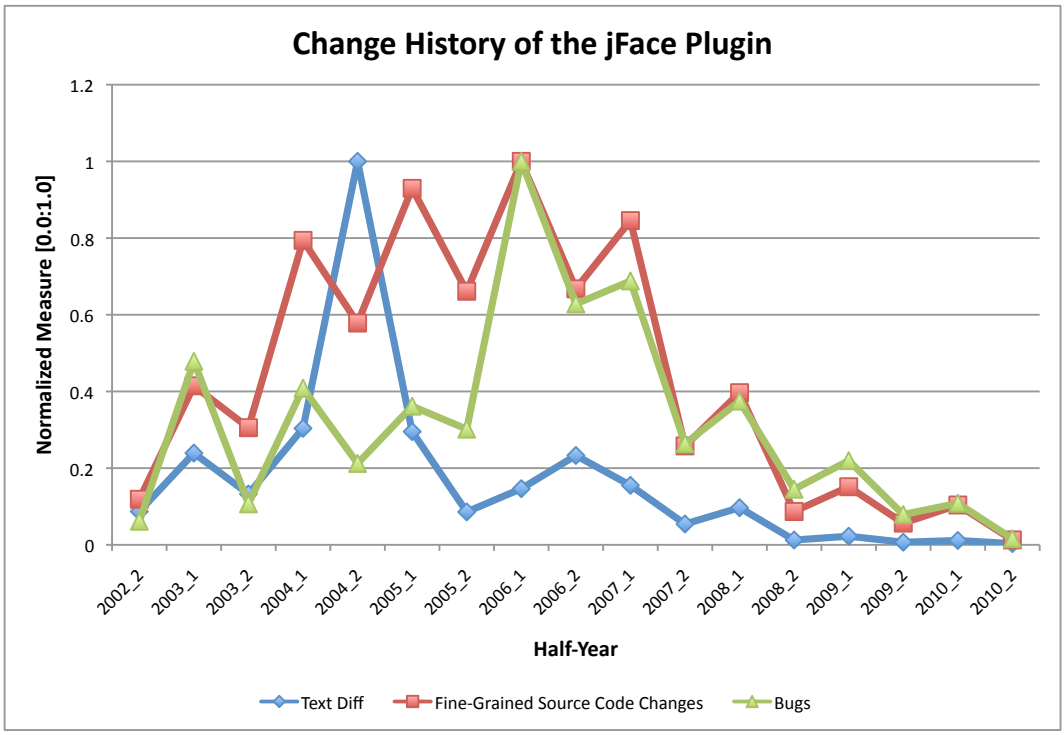


Figure 1.4: This figure shows the amount of lines changed (Text Diff) and AST based fine-grained source code chances as well as the number of bugs of the Eclipse jFace plugin per half-year (all numbers are normalized between [0.0:1.0].

In particular, our thesis is:

Using fine-grained source code changes allows us to build prediction models that take into account the semantics of changes and are more accurate in terms of prediction performance and prediction granularity.

1.4 Foundation and Structure of the Thesis

The foundation and contributions of this thesis consist of five empirical studies each containing a series of prediction experiments. We quantitatively study prediction models that are based on (fine-grained) source code changes and bug data extracted from the development history of several popular open-source projects, *e.g.*, Eclipse[5]. The selected projects are well known in the field of Mining Software Repositories and have been subject to numerous studies before. In particular, many studies on bug prediction models rely on their change and bug data.

The key focus of our empirical studies is on how software prediction models can benefit from measuring code changes at a fine-grained level and from the type semantics of changes. For that, a set of research hypotheses is formulated in each study.

In the remainder of this section we summarize for each empirical study (1) how the research hypotheses are embedded in this thesis, (2) how they add to the overall thesis statement (see Section 1.3), and (3) how they advance the current state of the art, *i.e.*, the contributions. Furthermore, we briefly give an overview of the results of the prediction experiments of each individual study and discuss their implications. The complete studies themselves are presented in Chapter 2–6 (see Section 1.5).

Study 1 *Comparing Fine-Grained Source Code Changes And Code Churn For Bug Prediction*: Code churn in terms of lines modified (*LM*) and past changes turned out to be significant indicators of bugs. However, as discussed in Section 1.3 these measures are rather imprecise and do not reflect all the detailed changes of particular source code entities during maintenance activities. Moreover, those change measures suffer from the fact that they do not capture the semantics of code changes.

[5]http://www.eclipse.org

For example, the file `BinaryCompareViewerCreator.java` of the Eclipse plugin Compare had 8 revisions and in total 81 lines were changed. None of these changes affected any source code entity since only license header or indentation updates have been performed. Between revision 1.1 and 1.2 of the source file `CompareEditorSelectionProvider.java` of the same plugin a single if-statement was removed. Between revisions 1.2 and 1.3 a nested if-statement was added, import statements were updated, and two methods were added.

This study is the first step at verifying our thesis statement, *i.e.*, improving bug prediction models using fine-grained source code changes (*SCC*). We explore with a series of prediction experiments using data from the Eclipse platform how *SCC* relates to bugs and to what extent bug prediction models benefit from having more detailed information about source code changes. The research hypotheses (**H**) of the study are:

H 1: *SCC* exhibit a stronger correlation with the number of bugs than *LM*.

H 2: *SCC* achieves better performance to classify source files into *bug-prone* and *not bug-prone*.

H 3: *SCC* achieves better performance when predicting the number of bugs in source files than *LM*.

Results of Study 1 The results can be summarized as follows: *SCC* outperforms *LM* for learning bug prediction models to classify source files into bug- and not bug-prone, as well as to predict the number of bugs in source files. In particular, with an average AUC value of 0.9 compared to 0.85 *SCC* achieved a significantly better classification performance than *LM*. Moreover, SCC improved the number of successful pair-wise cross-project prediction runs to 10% (out of 210 runs) compared to 1% obtained with LM. However, experiments with different binning

cut-points showed that the better classification performance of models computed with *SCC* diminishes as the predictive target class becomes smaller. Asymptotic regression models using *SCC* as independent variables exhibited significantly better explanatory power than the same models based on *LM*, *i.e.*, median R^2 of 0.79 versus 0.7.

An analysis based on the Information Gain criteria showed that a subset of all change types exhibit a relatively large predictive power. This leads to more compact prediction models that explicitly quantify the relation between a certain type of source code change and bugs.

Study 2 *Method-Level Bug Prediction*: Most of the current bug prediction approaches predict bugs at the level of source files (or binaries, modules, Java packages). However, since a file can be arbitrarily large, a developer needs to invest a significant amount of time to examine all methods of a file in order to locate a particular bug. Moreover, considering that larger files are known to be among the most bug-prone, the effort required for code inspection and review is even larger.

While we used fine-grained change data in **Study 1** to build more accurate prediction models in terms of prediction performance, we leverage fine-grained source code changes in the experiments of this study to predict bugs at a finer granularity, *i.e.*, at method-level. In particular, the hypothesis of **Study 2** is:

> **Hypothesis**: The concept of fine-grained source code changes enables predicting bugs at method-level.

We investigate the above hypothesis based on the source code and change history of 21 Java open-source projects.

Results of Study 2 The results of this study show that we can build prediction models that identify bug-prone methods with precision, recall,

and AUC of 0.84, 0.88, and 0.95, respectively. Furthermore, our experiments indicate that change metrics significantly outperform source code metrics for method-level bug prediction.

We argue that being able to narrow down the location of bugs to method-level can save manual inspection steps and significantly improve testing effort allocation. This is especially important in the omnipresent case of limited quality assurance resources.

For instance, in our dataset a class has on average 11 methods out of which 4 (~32%) are bug-prone, *i.e.*, are affected by at least one bug. Assuming that there is only knowledge that a file is bug-prone, but not which particular method contains the bug—as given by a file-level prediction model—a developer needs to inspect all methods one by one until the bug is located. Given the median precision of 0.84 achieved by one of our method-level based prediction models, a developer has roughly the same chance of picking a bug-prone method by randomly guessing after "eliminating" 6 out of those 11 methods ($4/5 = 0.8$).

In other words, one needs to manually reduce the set of possible candidates by more than half of all methods until chance is as good as our prediction models in terms of retrieving a bug-prone method.

Study 3 *Using the Gini Coefficient for Bug Prediction*: Prior work showed that not only properties of the source code itself, *e.g.*, size and complexity, and the development process, *e.g.*, change frequency, but also the social context of the development process affects the quality of a system (see Section 1.2.3). For instance, the number of authors and the contribution structure, *i.e.*, who modified a certain part of the source code, are related to bugs.

In this study we analyze the relationship between code ownership and bugs in source files. We understand code ownership by the fact that a relatively small subgroup of developers accumulates a major share of all changes done to a system (or to parts of it). For that, we apply the Gini

coefficient [Gin12]—a well known measure in the field of economy to express the disparity of a good's distribution among the individuals of a population. Analogously we measure how source code changes are distributed among developers. For instance, a high Gini coefficient for a given source code file means that a relatively small group of developers is responsible for a large amount of changes, *i.e.*, there is a high degree of code ownership for that file with respect to changes. We formulate and investigate the following research hypotheses in this study:

> **H 1**: The Gini coefficient based on change data correlates negatively with the number of bugs.
> **H 2**: The Gini coefficient based on change data can classify source files into *bug-prone* and *not bug-prone* files.

These hypotheses are motivated by the rationale that when a few developers contribute a major portion of all changes possibly less bugs occur as there is a clear responsibility and ownership. Whereas the case of the "too many cooks-situation" results in more uncoordinated, fragmented, and bug-prone changes.

Results of Study 3 The results using change data from the Eclipse development history suggest that focusing code changes on a relatively small group of dedicated developers is beneficial with respect to bugs. Moreover, based on the prediction models from **H 2** we can output a certain threshold of the Gini coefficient that should be kept as otherwise the likeliness of a file to be bug-prone rises above, for example, 90%.

Study 5 *Predicting the Types of Code Changes*: The results of the previous studies showed that past changes, especially, fine-grained source code changes (see **Study 1** & **2**), as well as the degree of the distribution of the code changes among the developers (see **Study 3**) are indicators of bugs.

Therefore, to better cope with software maintenance, in particular with changes, researchers investigated the use of source code metrics to train prediction models, which can guide developers towards the change-prone parts of a software system. The main motivation for these approaches is that developers can better focus on these change-prone parts in order to take appropriate counter measures to minimize the number of future changes, and consequently, minimize the number of future bugs. However, similarly to bug-prediction models, most of those approaches measure, and hence, predict changes in terms of revisions and textual differences. For instance, they do not provide detailed information whether a method invocation has been added, a condition expression of an if-statement has changed, or the type of a method parameter has changed.

We explore to which extent data-mining models can predict if a source file will be affected by a certain category of source code change types, *e.g.*, declaration changes. To compute those prediction models we focus on object-oriented metrics (OOM) and centrality measures from social network analysis (SNA) computed on the static source code dependency graph. The research hypotheses of this study are:

H 1: *OOM* and *SNA* measures correlate positively with fine-grained source code changes.
H 2: *OOM* and *SNA* measures can predict categories of source code changes.

Being able to predict not only if a file will most likely be affected by changes but additionally by what types of changes has practical benefits. For example, if a developer is made aware that there will be API changes, *i.e.*, declaration changes, she can plan accordingly and allocate resources for systemwide integration tests with dependent modules and, furthermore, she might account for additional time to update the API and design documents, and to synchronize with other developers using

the particular API. In contrast, if only small statement changes are predicted localized unit tests will be sufficient and no further change impact can be expected.

Results of Study 4 The results of our prediction experiments in this study show that OOM and SNA metrics can be used to compute models to predict the likelihood that a source file will be affected by a certain category of source code changes. For instance, the models for predicting changes in the method declarations of Java classes obtained a median precision of 0.82 and a recall of 0.77.

In all our models, the complexity of classes as well as the number of outgoing method invocations showed the highest correlation and predictive power for our change type categories. This finding was sustained by a manual analysis of a subset of changes.

Study 5 *Classifying Fast and Slowly Fixed Bugs in Open Source Projects*: Once a bug has been found or reported it needs to be fixed. However, the huge number of bugs that are reported for a large software system raises the problem that not enough resources and time are available to fix all existing bugs. Therefore, two important questions concerning the coordination of development effort is which bugs to fix first and how long it takes to fix them. This is analogous to the situation that drives and motivates bug prediction models as there are limited resources to test the entire system.

In this study we investigate prediction models which support developers in the cost/benefit analysis by giving recommendations which bugs should be fixed first. We address the question whether we can classify incoming bug reports into fast and slowly fixed. In particular, we investigate whether certain properties of a newly reported bug have an effect on how long it takes to fix that bug. For instance, intuitively one would expect that some of the properties, such as `priority` or `severity`,

have a significant influence on the fix time of a bug. Moreover, we analyze whether prediction models can be improved by including post-submission information within 1 to 30 days after a bug was reported. The two research hypotheses are:

H 1: Incoming bug reports can be classified into fast and slowly fixed.
H 2: Post-submission data of bug reports improves prediction models.

We investigate these two hypotheses with bug report data of six software systems taken from the three open source projects Eclipse, Mozilla, and Gnome. Decision tree analysis with 10-fold cross validation is used to train and test prediction models.

Results of Study 5 Summarized, the results of our experiments are: Between 60% and 70% of incoming bug reports can be correctly classified into fast and slowly fixed. Information regarding the person being responsible for a particular bug, about who reported the bug, and concerning the date a bug was opened are those attributes that have the strongest influence on the fix-time of bugs. Post-submission data of bug reports improves the performance of prediction models by 5% to 10%. The best-performing prediction models were obtained with 14-days or 30-days of post-submission data. The addition of concrete milestone information was the main factor for the performance improvements.

In summary, as the key contribution of this thesis we show how software prediction models and, in particular, bug prediction models could benefit from using fine-grained source code changes.

1.5 Thesis Roadmap

The remainder of this thesis is composed of Chapters 2–7. Chapters 2–6 are based on the empirical studies that frame the foundation of the thesis (see Section 1.4).

In Chapter 2 (p.29 *et seq.*) we compare fine-grained source code changes and code churn for bug prediction (**Study 1**). We then train data mining models with fine-grained source code changes in Chapter 3 (p.77 *et seq.*) to predict bugs on method-level instead of file-level (**Study 2**). Chapter 4 (p.105 *et seq.*) uses the Gini coefficient to measure code ownership and relates this measure to the bug-proneness of source code files (**Study 3**). In Chapter 5 (p.121 *et seq.*) we then investigate the extent to which the type of source code changes can be predicted (**Study 4**). Chapter 6 (p.147 *et seq.*) presents an approach to discriminate between slowly and fast fixed bugs based on the attributes of a bug report (**Study 5**). Figure 1.5 shows the arrangement and the relation between the empirical studies, the individual chapters of the thesis, and their associated publications. Each colored box denotes the key subject of an empirical study and its publication.

Chapter 7 (p.173 *et seq.*) concludes the thesis, discusses the implications of our findings, and highlights possibilities for future work.

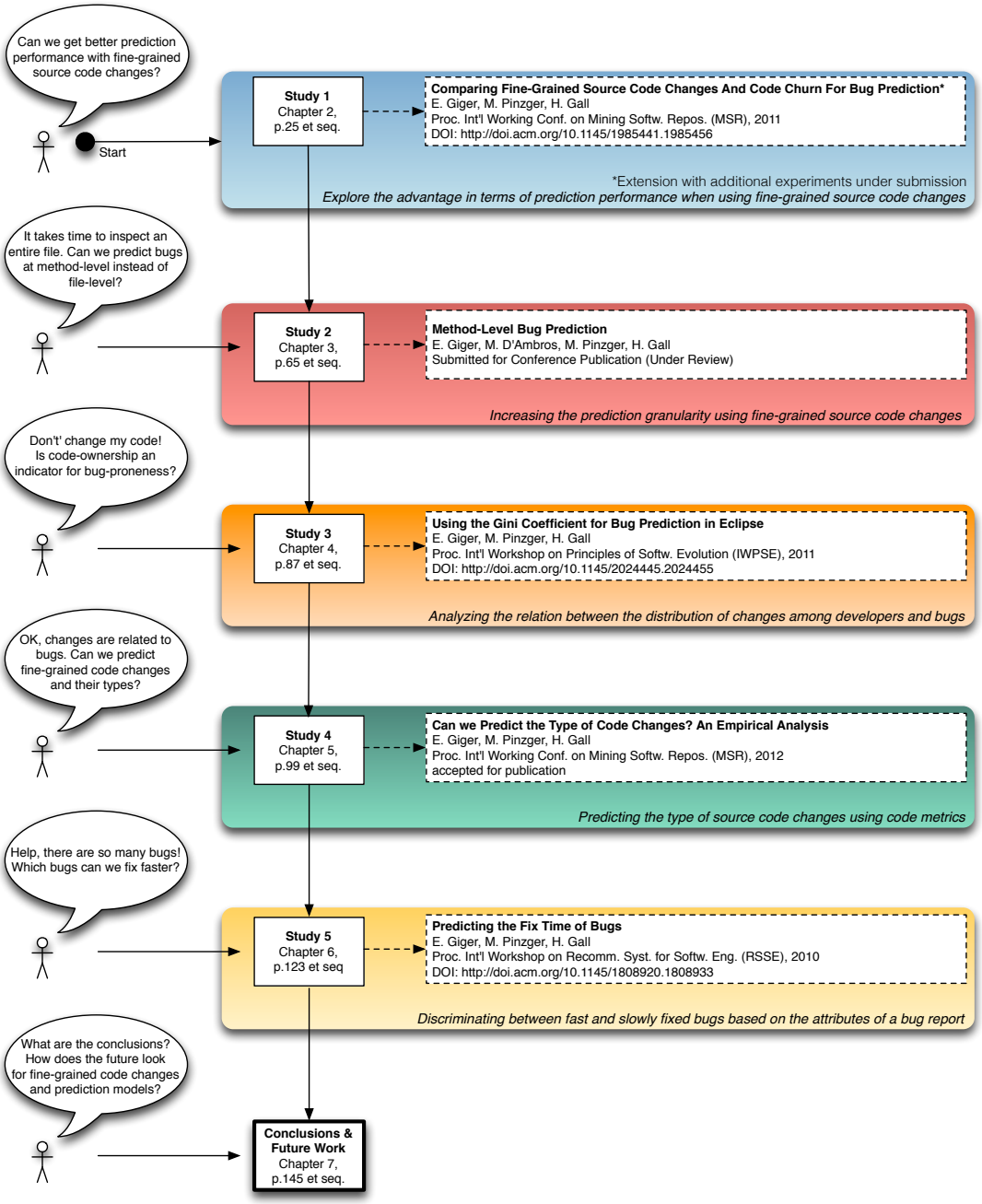


Figure 1.5: Thesis Roadmap: Relation between the empirical studies, the individual chapters, and their associated publications. Each colored box denotes the key subject of an empirical study and its publication.

2

Comparing Fine-Grained Source Code Changes And Code Churn For Bug Prediction

Comparing Fine-Grained Source Code Changes And Code Churn For Bug Prediction
E. Giger, M. Pinzger, H. Gall
Proc. Int'l Working Conf. on Mining Softw. Repos. (MSR), 2011
DOI: http://doi.acm.org/10.1145/1985441.1985456

Abstract

A significant amount of research effort has been dedicated to learning prediction models that allow project managers to efficiently allocate resources to those parts of a software system that most likely are bug-prone and therefore critical. Prominent measures for building bug prediction models are product measures, e.g., complexity or process measures, such as code churn. Code churn in terms of lines modified (*LM*) and past changes turned out to be significant indicators of bugs. However, these measures are rather imprecise and do not reflect all the detailed changes of particular source code entities during maintenance activ-

ities. In this paper, we explore the advantage of using *fine-grained source code changes* (*SCC*) for bug prediction. *SCC* captures the exact code changes and their semantics down to statement level. We present a series of experiments using different machine learning algorithms with a dataset from the Eclipse platform to empirically evaluate the performance of *SCC* and *LM*. The results show that *SCC* outperforms *LM* for learning bug prediction models.

2.1 Introduction

Bugs in software systems are a key risk and major cost driver for both, companies that develop software and companies that consume software systems in their daily business. Development teams are typically exposed to time pressure and costs. Often, Quality Assurance (QA) suffers from these constraints, and project managers are forced to allocate their limited resources with maximum efficiency. Research has developed bug prediction models that help managers in a structured manner to allocate QA resources to those parts of a system that likely contain most of the bugs rather than relying solely on their experience.

Prominent measures for building bug prediction models are product measures, *e.g.*, complexity [NBZ06], or process measures, such as code churn [GKMS00, NB05]. Prior work found out that process measures perform explicitly well [MPS08]. In their models, source files with a high code churn are most likely to be bug prone. However, existing measures such as code churn based on lines modified (*LM*) suffer from the fact that they do not capture the semantics of code changes. For example, the source file `BinaryCompare-ViewerCreator.java` of the Eclipse plugin Compare had 8 revisions and in total 81 lines were changed. None of these changes affected any source code entity since only license header or indentation updates have been performed. *Fine-grained source code changes* (*SCC*) as introduced by [FWPG07] on the other hand capture the semantics of changes. For example, between revision 1.1 and 1.2 of the source file `CompareEditorSelectionProvider.java` of the same plugin a single if-statement was removed. Between revisions 1.2 and 1.3 a nested if-statement was added, import statements were updated, and two methods were added.

In our previous work, we pointed out this discrepancy between changes based on a text-line level and fine-grained source code changes, and showed that fine-grained source code changes can be used to qualify change couplings between source files [FG06].

In this paper, we explore with a series of prediction experiments using data from the Eclipse platform how *SCC* relates to bugs and to what extent

bug prediction models benefit from having more detailed information about source code changes. In particular, we investigate the following three research hypotheses:

H 1: *SCC* exhibit a stronger correlation with the number of bugs than *LM*.

H 2: *SCC* achieves better performance to classify source files into *bug-* and *not bug-prone*.

H 3: *SCC* achieves better performance when predicting the number of bugs in source files than *LM*.

The results of our study with Eclipse projects show that *SCC* significantly outperforms *LM* for learning bug prediction models to classify source files into *bug-* and *not bug-prone*, as well as to predict the number of bugs in source files. Furthermore, an analysis based on *Information Gain* showed that a subset of all change types exhibit relatively large predictive power and can be used to train more compact prediction models.

The remainder of this paper is organized as follows: In Section 2.2, we give an overview of our approach and outline the steps necessary to prepare the data. Section 2.3 presents the empirical study with the Eclipse projects. We discuss our findings in Section 2.4 and threats to validity in Section 2.5. In Section 2.6, we present related work and then draw our conclusions in Section 2.7.

2.2 Approach

In this section, we describe the methods and tools we used to extract and preprocess the data (see Figure 2.1). Basically, we take into account three main pieces of information about the history of a software system to assemble the dataset for our experiments: (1) versioning data including lines modified (*LM*),

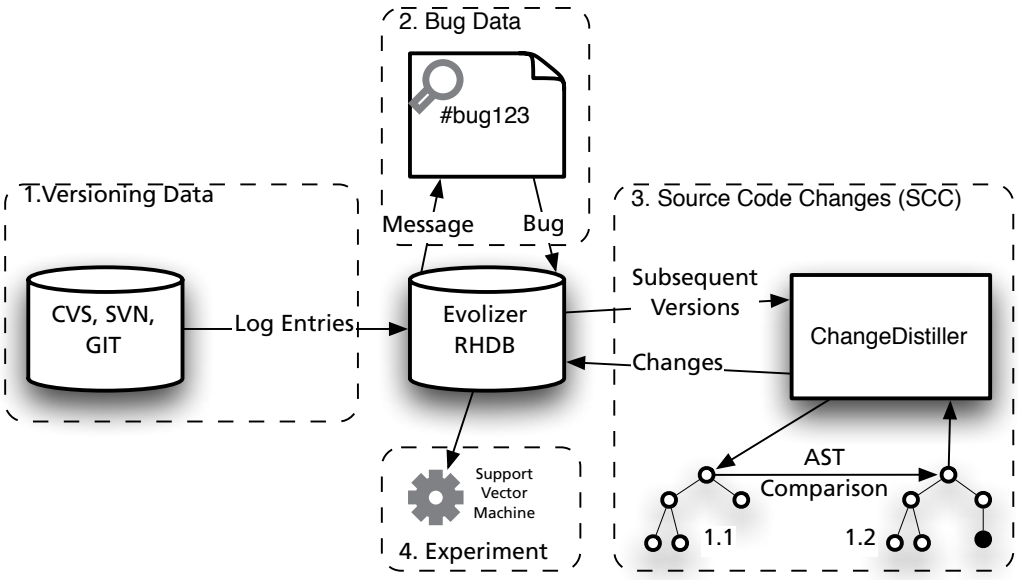


Figure 2.1: Overview of the main phases of the data extraction process.

(2) bug data, *i.e.*, which files contained bugs and how many of them (*Bugs*), and (3) fine-grained source code changes (*SCC*).

1. Versioning Data. We use EVOLIZER [GFP09] to access the versioning repositories, *e.g.*, CVS, SVN, or GIT. They provide log entries that contain information about revisions of files that belong to a system. From the log entries we extract the revision number (to identify the revisions of a file in correct temporal order), the revision timestamp, the name of the developer who checked-in the new revision, and the commit message. We then compute *LM* for a source file as the sum of lines added, lines deleted, and lines changed per file revision.

2. Bug Data. Bug reports are stored in bug repositories such as Bugzilla. Traditional bug tracking and versioning repositories are not directly linked. We first establish these links by searching references to reports within commit messages, *e.g.*,"fix for 12023" or "bug#23467". Prior work used this method and developed advanced matching patterns to catch those references [ZPZ07, SZZ05, FPG03]. Again, we use EVOLIZER to automate this process. We take

into account all references to bug reports. Based on the links we then count the number of bugs (*Bugs*) per file revision.

3. Fine-Grained Source Code Changes (SCC): Current versioning systems record changes solely on file level and textual basis, *i.e.*, source files are treated as pure text files. In [FG06], Fluri *et al.* showed that *LM* recorded by versioning systems might not accurately reflect changes in the source code. For instance, source formatting or license header updates generate additional *LM* although no source code entities were changed. Moreover, *LM* does not convey any information about the *type* of a change. Changing the name of a local variable and a method likely result both in "1 line changed" but are different modifications. Fluri *et al.* developed a tree differencing algorithm for fine-grained source code change extraction [FWPG07]. It allows to track fine-grained source code changes down to the level of single source code statements, *e.g.*, method invocation statements, between two versions of a source file by comparing their respective abstract syntax trees (AST).

This comparison requires two steps: First, matching nodes between two AST versions are detected using string similarity measures for leave nodes and tree similarity measures regarding subtrees. Finding such node matches between two versions of an AST is necessary to determine wether a node was inserted, deleted, update, moved, or did not experience any change at all. *Inserting*, *deleting*, *updating*, and *moving* nodes constitute the basic tree edit operations that can possibly alter the structure of an AST (or any tree like structure in general) [FWPG07].

Second, a (minimal) set of such basic tree edit operations transforming one version of the AST into the other is extracted. Each tree edit operation for a given node is then combined with the semantic information about the source code entity that this node represents within the AST. This allows to classify a basic tree edit operation using the *taxonomy of source code changes* presented in [FG06]. For instance, consider two AST nodes, A and B, that were inserted. A represents a method invocation statement within the AST structure and B an else-part. Accordingly, these two basic insert operations are classified as *statement insert* and *else-part insert* respectively.

The algorithm is implemented in CHANGEDISTILLER [GFP09] that pairwise compares the ASTs between all direct subsequent revisions of each file. Based on this information, we then count the number of different source code changes (*SCC*) per file revision.

The preprocessed data from step 1-3 is stored into the Release History Database (RHDB) [FPG03]. From that data, we then compute *LM*, *SCC*, and *Bugs* for each source file by aggregating the values over the given observation period.

2.3 Empirical Study

In this section, we present the empirical study that we performed to investigate the hypotheses stated in Section 2.1. We discuss the dataset, the statistical methods and machine learning algorithms we used, and report on the results and findings of the experiments.

2.3.1 Dataset and Data Preparation

We performed our experiments on 15 plugins of the Eclipse platform. Eclipse is a popular open source system that has been studied extensively before [BEP07, NZZ+10, ZPZ07, ZNG+09].

Table 2.1 gives an overview of the Eclipse dataset used in this study with the number of unique **.java* files (Files), the total number of java file revisions (Rev.), the total number of lines added, deleted, and changed (*LM*), the total number of fine-grained source code changes (SCC), and the total number of bugs (*Bugs*) within the given time period (Time). Only source code files, *i.e.*, **.java*, are considered.

After the data preparation step, we performed an initial analysis of the extracted *SCC*. This analysis showed that there are large differences of change type frequencies, which might influence the results of our empirical study. For instance, the change types *Parent Class Delete*, *i.e.*, removing a super class

Table 2.1: Eclipse dataset used in this study.

Eclipse Project	Files	Rev.	*LM*	*SCC*	*Bugs*	Time
Compare	278	3'736	140'784	21'137	665	May01–Sep10
jFace	541	6'603	321582	25'314	1'591	Sep02–Sep10
JDT Debug	713	8'252	218'982	32'872	1'019	May01–July10
Resource	449	7'932	315'752	33'019	1'156	May01–Sep10
Runtime	391	5'585	243'863	30'554	844	May01–Jun10
Team Core	486	3'783	101'913	8'083	492	Nov01–Aug10
CVS Core	381	6'847	213'401	29'032	901	Nov01–Aug10
Debug Core	336	3'709	85'943	14'079	596	May01–Sep10
jFace Text	430	5'570	116'534	25'397	856	Sep02–Oct10
Update Core	595	8'496	251'434	36'151	532	Oct01–Jun10
Debug UI	1'954	18'862	444'061	81'836	3'120	May01–Oct10
JDT Debug UI	775	8'663	168'598	45'645	2'002	Nov01–Sep10
Help	598	3'658	66'743	12'170	243	May01–May10
JDT Core	1'705	63'038	2'814K	451'483	6'033	Jun01–Sep10
OSGI	748	9'866	335'253	56'238	1'411	Nov03–Oct10

from a class declaration, or *Removing Method Overridability*, *i.e.*, adding the java keyword `final` to a method declaration, are relatively rare change types. They constitute less than one thousandth of all *SCC* in the entire study corpus. Whereas one fourth of all *SCC* are *Statement Insert* changes, *e.g.*, the insertion of a new local variable declaration.

We therefore aggregate *SCC* according to their change type semantics into 7 categories of *SCC* for our further analysis. Table 2.2 shows the resulting aggregated categories and their respective meanings.

Some change types defined in [FG06] such as the ones that change the declaration of an attribute are left out in our analysis as their total frequency is below 0.8%. The complete list of all change types, their meanings and their contexts can be found in [FG06].

Table 2.3 shows the relative frequencies of each change type category per Eclipse project, plus their mean, variance, and the 95% confidence interval over all selected projects. Looking at the mean values listed in the second last row of the table, we can see that 70% of all changes are *stmt* changes. These are

Table 2.2: Categories of fine-grained source code changes

Category	Description
cDecl	Aggregates all changes that alter the declaration of a class: Modifier changes, class renaming, class API changes, parent class changes, and changes in the "`implements` list".
oState	Aggregates the insertion and deletion of object states of a class, *i.e.*, adding and removing fields.
func	Aggregates the insertion and deletion of functionality of a class, *i.e.*, adding and removing methods.
mDecl	Aggregates all changes that alter the declaration of a method: Modifier changes, method renaming, method API changes, return type changes, and changes of the parameter list.
stmt	Aggregates all changes that modify executable statements, *e.g.*, insertion or deletion of statements.
cond	Aggregates all changes that alter condition expressions in control structures.
else	Aggregates the insertion and deletion of *else*-parts.

relatively small changes and affect only single statements. Changes that affect the existing control flow structures, *i.e.*, *cond* and *else*, constitute only about 6% on average. While these changes might affect the behavior of the code, their impact is locally limited to their proximate context and blocks. They ideally do not induce changes at other locations in the source code. *cDecl*, *oState*, *func*, and *mDecl* represent about one fourth of all changes in total. They change the interface of a class or a method and do—except when adding a field or a method—require a change in the dependent classes and methods. The impact of these changes is according to the given access modifiers; within the same class or package (`private` or `default`) or external code (`protected` or `public`).

The values in Table 2.3 show small variances and relatively narrow confidence intervals among the categories across all projects. This is an interesting

Table 2.3: Relative frequencies of *SCC* categories per Eclipse project, plus their mean, variance, and 95% confidence interval (CI) over all selected projects.

Eclipse Project	cDecl	oState	func	mDecl	stmt	cond	else
Compare	0.01	0.06	0.08	0.05	0.74	0.03	0.03
jFace	0.02	0.04	0.08	0.11	0.70	0.02	0.03
JDT Debug	0.02	0.06	0.08	0.10	0.70	0.02	0.02
Resource	0.01	0.04	0.02	0.11	0.77	0.03	0.02
Runtime	0.01	0.05	0.07	0.10	0.73	0.03	0.01
Team Core	0.05	0.04	0.13	0.17	0.57	0.02	0.02
CVS Core	0.01	0.04	0.10	0.07	0.73	0.02	0.03
Debug Core	0.04	0.07	0.02	0.13	0.69	0.02	0.03
jFace Text	0.04	0.03	0.06	0.11	0.70	0.03	0.03
Update Core	0.02	0.04	0.07	0.09	0.74	0.02	0.02
Debug UI	0.02	0.06	0.09	0.07	0.70	0.03	0.03
JDT Debug UI	0.01	0.07	0.07	0.05	0.75	0.02	0.03
Help	0.02	0.05	0.08	0.07	0.73	0.02	0.03
JDT Core	0.00	0.03	0.03	0.05	0.80	0.05	0.04
OSGI	0.03	0.04	0.06	0.11	0.71	0.03	0.02
Mean	0.02	0.05	0.07	0.09	0.72	0.03	0.03
Variance	.000	.000	.001	.001	.003	.000	.000
95% CI Mean	.01–.02	.04–.05	.06–.08	.06–.08	.63–.68	.018–.026	.019–.027

observation as the selected Eclipse projects do vary in terms of number of files and changes (see Table 2.1).

2.3.2 Correlation of SCC Categories

We first performed a correlation analysis between the different SCC categories of all source files of the selected projects. We use the Spearman rank correlation because it makes no assumptions about the distributions, variances and the type of relationship. It compares the ordered ranks of the variables to measure a monotonic relationship. This makes Spearman more robust than Pearson correlation, which is restricted to measure the strength of a linear association between two normal distributed variables [DWC04]. Spearman values of +1 and -1 indicate a high positive or negative correlation, whereas 0 tells that the

variables do not correlate at all. Values greater than +0.5 and lower than -0.5 are considered to be substantial; values greater than +0.7 and lower than -0.7 are considered to be strong correlations [PNM08].

Table 2.4 lists the results. Some facts can be read from the values: *cDecl* does neither have substantial nor strong correlation with any of the other change types. *oState* has its highest correlation with *func*. *func* has approximately equal high correlations with *oState*, *mDecl*, and *stmt*. The strongest correlations are between *stmt*, *cond*, and *else* with 0.71, 0.7, and 0.67.

While this correlation analysis helps to gain knowledge about the nature and relation of change type categories it mainly reveals multicollinearity between those categories that we have to address when building regression models. A causal interpretation of the correlation values is tedious and must be dealt with caution. Some correlations make sense and could be explained using common knowledge about programming. For instance, the strong correlations between *stmt*, *cond*, and *else* can be explained by local variables that often are affected when existing control structures are changed. This is because they might be moved into a new *else*-part or because a new local variable is needed to handle the different conditions. In [FGG08], Fluri *et al.* attempt to find an explanation why certain change types occur more frequently together than others, *i.e.*, why they correlate. Their results show that some change types are more often applied together when they are part of a change type pattern, *e.g.*, consistenly introducing the *single exit* principle in an existing code base.

2.3.3 Correlation of Bugs, LM, and SCC

H 1 formulated in Section 2.1 aims at analyzing the correlation between *Bugs*, *LM*, and *SCC* (on the level of source files). It serves two purposes: (1) We analyze whether there is a significant correlation between *SCC* and *Bugs*. A significant correlation is a precondition for any further analysis and prediction model. (2) Prior work reported on the positive relation between *Bugs* an *LM*. We explore the extent to which *SCC* has a stronger correlation with *Bugs* than *LM*. We apply the Spearman rank correlation to each selected Eclipse project

Table 2.4: Spearman rank correlation between *SCC* categories (*marks significant correlations at $\alpha = 0.01$.)

	cDecl	oState	func	mDecl	stmt	cond	else
cDecl	1.00*	0.33*	0.42*	0.49*	0.23*	0.21*	0.21*
oState		1.00*	0.65*	0.53*	0.62*	0.51*	0.51*
func			1.00*	0.67*	0.66*	0.53*	0.53*
mDecl				1.00*	0.59*	0.49*	0.48*
stmt					1.00*	0.71*	0.7*
cond						1.00*	0.67*
else							1.00*

to investigate **H 1**.

Table 2.5 lists the results of the correlation analysis per project. The second and third columns on the left hand side show the correlation values between *Bugs* and *LM*, and total *SCC*. The values for *LM* show that except for two projects all correlations are at least substantial, some are even strong. The mean of the correlation is 0.62 and the median is 0.66. This indicates that there is a substantial, observable positive correlation between *LM* and bugs meaning that an increase in *LM* leads to an increase in bugs in a source file. This result confirms previous research presented in [GKMS00, NB05, NZZ+10].

The values in the third column show that all correlations for *SCC* are positive and most of them are strong. The mean of the correlation is 0.74 and the median is 0.77. Some Eclipse projects show correlation values of 0.8 and higher. Two values are below 0.7 and only one is slightly lower than 0.5. All values are statistically significant. This denotes an overall strong correlation between *Bugs* and *SCC* that is even stronger than between *Bugs* and *LM*. We applied a *One Sample Wilcoxon Signed-Ranks Test* on the *SCC* correlation values against the hypothesized limits of 0.5< (substantial) and 0.7< (strong). They were significant at $\alpha = 0.05$. Therefore we conclude that there is a significant strong correlation between *Bugs* and *SCC*.

We further compared the correlation values of *LM* and *SCC* in Table 2.5 to test whether the observed difference is significant. On average, the correlation between *Bugs* and *SCC* is 0.12 stronger than the correlation between *Bugs* and

Table 2.5: Spearman rank correlation between *Bugs* and *LM*, *SCC*, and *SCC* categories (*marks significant correlations at $\alpha = 0.01$).

Eclipse Project	*LM*	*SCC*	cDecl	oState	func	mDecl	stmt	cond	else
Compare	0.68*	**0.76***	0.54*	0.61*	0.67*	0.61*	0.66*	0.55*	0.52*
jFace	**0.74***	0.71*	0.41*	0.47*	0.57*	0.63*	0.66*	0.51*	0.48*
Resource	0.75*	**0.86***	0.49*	0.62*	0.70*	0.73*	0.67*	0.49*	0.46*
Team Core	0.15*	**0.66***	0.44*	0.43*	0.56*	0.52*	0.53*	0.36*	0.35*
CVS Core	0.60*	**0.79***	0.39*	0.62*	0.66*	0.57*	0.72*	0.58*	0.56*
Debug Core	0.63*	**0.78***	0.45*	0.55*	0.61*	0.51*	0.59*	0.45*	0.46*
Runtime	0.66*	**0.79***	0.47*	0.58*	0.66*	0.61*	0.66*	0.55*	0.45*
JDT Debug	0.62*	**0.80***	0.42*	0.45*	0.56*	0.55*	0.64*	0.46*	0.44*
jFace Text	**0.75***	0.74*	0.50*	0.55*	0.54*	0.64*	0.62*	0.59*	0.55*
JDT Debug UI	0.80*	**0.81***	0.46*	0.57*	0.62*	0.53*	0.74*	0.57*	0.54*
Update Core	0.43*	**0.62***	0.63*	0.40*	0.43*	0.51*	0.45*	0.38*	0.39*
Debug UI	0.56*	**0.81***	0.44*	0.50*	0.63*	0.60*	0.72*	0.54*	0.52*
Help	**0.54***	0.48*	0.37*	0.43*	0.42*	0.43*	0.44*	0.36*	0.41*
JDT Core	0.70*	**0.74***	0.39*	0.60*	0.69*	0.70*	0.67*	0.62*	0.60*
OSGI	0.70*	**0.77***	0.47*	0.60*	0.66*	0.65*	0.63*	0.57*	0.48*
Mean	0.62	0.74	0.46	0.53	0.60	0.59	0.63	0.51	0.48
Median	0.66	0.77	0.45	0.55	0.62	0.60	0.66	0.54	0.48

LM. In particular, 12 out of 15 cases show a stronger correlation towards *SCC* with an average difference of 0.16. In some cases the differences are even more pronounced, *e.g.*, 0.51 for Team Core or 0.25 for Debug UI. Other projects experience smaller differences such as 0.01 for JDT Debug UI and jFace, and 0.04 for JDT Core. Only in three cases the correlation of *LM* is stronger. The largest difference is 0.06 for Eclipse Help.

We used a *Related Samples Wilcoxon Signed-Ranks Test* to test the significance of the correlation differences between *LM* and *SCC*. The rationale for such a test is that (1) we calculated both correlations for each project resulting in a matched correlation pair per project and (2) we can relax any assumption about the distribution of the values. The test was significant at $\alpha = 0.05$ rejecting the null hypothesis that the two medians are the same. Based on this result we can accept **H 1**—*SCC* does have a stronger correlation with *Bugs* than *LM*.

As part of investigating **H 1**, we also analyzed the correlation between *Bugs* and the *SCC* categories we have defined in Table 2.2 to answer the question whether there are differences in how change types correlate with bugs.

The columns 4–10 on the right hand side of Table 2.5 show the correlations between the different categories and bugs for each Eclipse project. Regarding their mean, the categories *stmt*, *func*, and *mDecl* show the strongest correlation with *Bugs*. For some projects their correlation values are close or above 0.7, *e.g.*, *func* for Resource or JDT Core; *mDecl* for Resource and JDT Core; *stmt* for JDT Debug UI and Debug UI. *oState* and *cond* still have a substantial correlation with the number of bugs indicated by an average correlation value of 0.53 and 0.51. *cDecl* and *else* have means below 0.5. This indicates that *SCC* categories do correlate differently with the number of bugs in our dataset.

To test whether this assumption holds, we first performed a *Related Samples Friedman Test*. The result was significant at $\alpha = 0.05$, so we can reject the null hypothesis that the distribution of the correlation values of *SCC* categories, *i.e.*, the rows on the right hand side in Table 2.5 are the same. The *Friedman Test* operates on the mean ranks of related groups. We used this test because we repeatedly measured the correlations of the different categories on the same dataset, *i.e.*, our related groups, and because it does not make any assumption about the distribution of the data and the sample size.

A *Related Samples Friedman Test* is a global test that only tests whether all of the groups differ. It does not tell anything between which groups the difference occurs. To test whether some pairwise groups differ stronger than others or do not differ at all post-hoc tests are required. We performed a *Wilcoxon Test* and *Friedman Test* on each pair including α-adjustment.

The results showed two groups of *SCC* categories whose correlation values are not significantly different among each other: (1) *else*, *cond*, *oState*, and *cDecl*, and (2) *stmt*, *func*, and *mDecl*. The difference of correlation values between these groups is significant.

In summary, we found strong positive correlation between *SCC* and *Bugs* that is significantly stronger than the correlation between *LM* and *Bugs*. This indicates that *SCC* exhibits good predictive power, therefore we accepted

H 1. Furthermore, we observed a difference in the correlation values between several *SCC* categories and *Bugs*.

2.3.4 Predicting Bug- & Not Bug-Prone Files

The goal of **H 2** is to analyze how *SCC* performs compared to *LM* when discriminating between *bug-prone* and *not bug-prone* files in our dataset. We built models based on different machine learning techniques (in the following also called classifiers) and evaluated them with our Eclipse dataset.

Prior work states that some machine learning techniques perform better than others. For instance, [LBSP08] found out with an extended set of various classifiers that *Random Forest* performs the best on a subset of the NASA Metrics dataset. But in return they state as well that performance differences between classifiers are marginal and not necessarily significant.

For that reason we used the following classifiers: *Logistic Regression* (LReg), *J48* (C4.5 Decision Tree), *RandomForest* (RFor), *Bayesian Network* (BNet) implemented by the WEKA toolkit [WF05], *Exhaustive CHAID*, a Decision Tree based on chi squared criterion by SPSS 18.0, *Support Vector Machine (LibSVM)* [CL01], *Naive Bayes Network* (NBayes) and *Neural Nets* (NN) both provided by the Rapid Miner toolkit [MWK+06]. The classifiers calculate and assign a probability to each source file to be classified either into *bug-prone* or *not bug-prone*.

For each Eclipse project, we binned files into *bug-prone* and *not bug-prone* using the median of the number of bugs per file:

$$bugClass = \begin{cases} not\ bug-prone & : \quad Bugs <= median \\ bug-prone & : \quad Bugs > median \end{cases} \tag{2.1}$$

When using the median as cut point the labeling of a file is relative to how much bugs other files have in a project. This resulted in an average 57:43 prior probability towards *not bug-prone* file in our dataset. There exist several ways of binning files afore. They mainly vary in that they result

in different prior probabilities: For instance [ZPZ07] and [BEP07] labeled files as *bug-prone* if they had at least one bug. When having heavily skewed distributions this approach may lead to a high prior probability towards one class. [NZZ+10] used a statistically lower confidence bound. The different prior probabilities make the use of accuracy as a performance measure for classification difficult. As proposed in [LBSP08, MGF07], we therefore use the *area under the receiver operating characteristic curve* (AUC) as performance measure. AUC is independent of prior probabilities and therefore a robust measure to asses and compare the performance of prediction models [BEP07]. AUC can be seen as the probability that a trained model assigns a higher score to the *bug-prone* file when choosing randomly a *bug-prone* and a *not bug-prone* file [GS66]. We mainly use AUC for discussing and comparing the performance of prediction models. In addition, we also report on precision (P) and recall (R) to facilitate the comparison with existing work.

We performed four experiments to investigate **H 2**: In **Experiment 1** (**E 1**), we use logistic regression once with total number of *LM* and once with number of *SCC* per file as predictors. In **Experiment 2** (**E 2**), we use the above mentioned classifiers and *SCC* categories as predictors to investigate whether the additional information about the change type category can improve the performance of classification models. **Experiment 3** (**E 3**) analyzes the extent to which it is possible to use only a subset of all change type categories for model building without a significant loss in classification performance. In **Experiment 4** (**E 4**) we compare the performance of *LM* and *SCC* in terms of cross-project prediction.

In the following we discuss the results of all four experiments mainly by means of the AUC measure.

Experiment 1: Table 2.6 lists the AUC values of **E 1** for each project in our dataset. The models were validated using 10 fold cross validation,[1] and the performance measures were computed when reapplying the prediction model

[1] All 10 fold cross validation procedures in this work use stratified sampling, *i.e.*, the class distribution is kept constant [MWK+06].

Table 2.6: AUC, precision, and recall of **E1** using logistic regression with *LM* and *SCC* to classify source files into *bug-prone* or *not bug-prone*.

Eclipse Project	AUC_{LM}	AUC_{SCC}	P_{LM}	P_{SCC}	R_{LM}	R_{SCC}
Compare	0.84	**0.85**	0.76	**0.78**	**0.88**	0.81
jFace	0.90	0.90	0.81	**0.83**	0.85	**0.87**
JDT Debug	0.83	**0.95**	0.79	**0.85**	0.71	**0.91**
Resource	0.87	**0.93**	0.75	**0.80**	0.85	**0.93**
Runtime	0.83	**0.91**	0.71	**0.85**	**0.89**	0.83
Team Core	0.62	**0.87**	0.48	**0.69**	0.73	**0.82**
CVS Core	0.80	**0.90**	0.78	**0.89**	0.78	**0.83**
Debug Core	0.86	**0.94**	0.68	**0.82**	**0.92**	0.91
jFace Text	0.87	0.87	**0.70**	0.67	**0.87**	0.86
Update Core	0.78	**0.85**	0.63	**0.72**	**0.91**	0.88
Debug UI	0.85	**0.93**	0.64	**0.76**	0.87	**0.91**
JDT Debug UI	0.90	**0.91**	0.76	**0.78**	**0.89**	0.87
Help	**0.75**	0.70	**0.75**	0.63	**0.69**	0.63
JDT Core	0.86	**0.87**	0.76	**0.77**	0.82	**0.83**
OSGI	0.88	0.88	0.80	**0.87**	**0.86**	0.81
Median	0.85	**0.90**	0.75	**0.78**	0.85	**0.86**
Overall	0.85	**0.89**	0.70	**0.74**	0.77	**0.86**

to the dataset it was obtained from. *Overall* denotes the AUC of the model that was learned when merging all files of the projects into one larger dataset.

SCC achieves a very good performance with a median of 0.90 (see column AUC_{SCC}). This means that logistic regression using *SCC* as predictor ranks *bug-prone* files higher than *not bug-prone* files with a probability of 90%. Even the Help project, that shows the lowest AUC value, is still within the range of 0.7 what [LBSP08] call "promising results" . This comparatively low value is accompanied with the smallest correlation of 0.48 between *SCC* and *Bugs* in Table 2.5. The good performance of logistic regression and *SCC* is confirmed by an overall AUC value of 0.89 when learning the model from the entire dataset. With a value of 0.004 AUC_{SCC} has a low variance across all projects indicating consistent prediction models.

With a median AUC of 0.85, *LM* shows a lower performance than *SCC* (see

Table 2.7: AUC of **E 2** using different classifiers with the *SCC* categories as predictors for *bug-prone* and *not bug-prone* files (AUC of the best performing classifier per project is printed in bold).

Eclipse Project	LReg	J48	RFor	BNet	eCHAID	LibSVM	NBayes	NN
Compare	0.82	0.77	0.77	**0.83**	0.74	0.81	0.82	0.82
jFace	0.90	0.85	0.88	0.89	0.83	**0.91**	0.87	0.88
JDT Debug	0.94	0.92	0.94	**0.95**	0.89	**0.95**	0.87	0.89
Resource	0.89	0.86	0.89	0.91	0.77	**0.92**	0.90	0.91
Runtime	**0.89**	0.82	0.83	0.87	0.80	0.87	0.86	0.87
Team Core	**0.86**	0.78	0.79	0.85	0.77	**0.86**	0.85	**0.86**
CVS Core	**0.89**	0.81	0.87	0.88	0.74	0.87	0.86	0.88
Debug Core	0.92	0.86	0.89	0.91	0.79	**0.93**	0.92	0.86
jFace Text	**0.86**	0.77	0.81	0.85	0.76	0.79	0.82	0.81
Update Core	0.82	0.87	**0.90**	0.86	0.86	0.89	0.89	0.90
Debug UI	**0.92**	0.88	0.91	**0.92**	0.82	**0.92**	0.89	0.91
JDT Debug UI	0.89	0.89	**0.90**	0.89	0.81	**0.90**	0.85	0.89
Help	**0.69**	0.65	0.67	**0.69**	0.63	**0.69**	**0.69**	0.68
JDT Core	0.85	0.86	0.88	**0.90**	0.80	0.88	0.85	0.87
OSGI	0.86	0.81	0.86	**0.88**	0.77	0.87	0.87	0.87
Median	**0.89**	0.85	0.88	0.88	0.79	0.88	0.86	0.87
Overall	0.88	0.87	0.84	**0.89**	0.82	**0.89**	0.85	0.84

column AUC_{LM}). Help is the only project where *LM* is a better predictor than *SCC*. This is not surprising as it is the project that yields the largest difference in correlation in favor of *LM* (see Table 2.5). In general, the correlation values in Table 2.5 reflect the picture given by the AUC values. For instance, jFace, jFace Text, and JDT Debug UI that exhibit similar correlations performed similarly. A *Related Samples Wilcoxon Signed-Ranks Test* on the AUC values of *LM* and *SCC* was significant at $\alpha = 0.05$: Logistic regression based on *SCC* is not only a good predictor but is a significant better predictor than *LM* to classify source files of Eclipse projects into *bug-prone* and *not bug-prone*. Therefore, we can accept **H 2**—*SCC* achieves better performance when discriminating between *bug-* and *not bug-prone* files than *LM*.

Experiment 2: Table 2.7 lists the AUC values of each classifier for each project in our dataset. Analogously to **E 1**, the values for AUC, precision, and recall

were computed when reapplying the prediction model to the dataset it was obtained from (we skip the values for precision and recall for readability and space reasons). As before, the models were validated using 10 fold cross validation. *Overall* denotes the AUC of the model that was learned when merging all files of the projects into one larger dataset. When using logistic regression, multicollinearity between multiple predictors (see Table 2.3) may compromise the validity of the resulting model [DWC04]. To avoid this problem, we applied principal component analysis (PCA) based on the covariance matrix and a variance threshold of 0.95. PCA extracted one component which has been used to perform the logistic regression.

The results in Table 2.7 show median AUC values of approximately 0.8 and higher, which indicates that all selected classifiers obtain models with adequate performance. Furthermore, we can observe that LibSVM is the best classifier for 8 projects. BNet obtains similarly good results: According to the AUC values it is the top classifier for 6 projects and has together with LibSVM the highest AUC value when learning the prediction models from the entire dataset. Not surprisingly, logistic regression also yields a good performance with a high median AUC of 0.89 which is similar to the result in **E 1** (the input from PCA accounts for more than 95% of the *SCC* in our dataset).

RFor and NN—though not the best—are still good classifiers and among the best for some projects. They fall slightly behind because of their performance on the overall dataset. The decision tree methods J48 and eCHAID show lower performance compared to the other classifiers. None of them performs best for one project. Furthermore, eCHAID has the lowest median AUC and performs the worst on the entire dataset.

Next, we compared the results of both experiments to find out whether including the information about the *SCC* category helps to improve the performance of prediction models. We compared the AUC values from LibSVM (the best performing classifier in **E 2**) with the AUC values from the logistic regression in **E 1** using the *WilcoxonTest*. The test was not significant, therefore we can conclude that the inclusion of the *SCC* category does not lead to better performing prediction models.

For the discussion of the performance differences between several classifiers we used a *Related Samples Friedman Test* and an adjusted α level for the post-hoc comparison of each classifier pair. The test was significant at $\alpha = 0.05$. This means that there is a statistically significant difference between the mean ranks of the AUC values. However, a look at the pairwise tests revealed that the significance is mainly due to the low performance of eCHAID and to some extent due to J48. The differences between the other pairs that did not involve a decision tree method were not significant. These results confirm the experience drawn in prior work: (1) There is a relatively good performance of more complex classifiers in our experiments, *e.g.*, LibSVM or RFor. But their performance does not differ statistically significant in most cases [LBSP08]; (2) the good performance of Bayesian methods [MGF07]; and (3) in particular the comparably good predicting power of SVM for Eclipse data [SZZ06].

Based on the AUC values in Table 2.6 and Table 2.7 we conclude that *SCC* (**E 1**) as well as their categories (**E 2**) are good predictors for *bug-prone* and *not bug-prone* files. *SCC* outperformed the prediction models built with *LM*, therefore we accept **H 2**.

Experiment 3: The results of the two previous experiments show that *SCC* are good predictors for *bug-prone* files and achieve better classification results than *LM*. Furthermore, the results of **E 2** indicated that including the information about the categories of change types did not lead to any significant differences regarding the performance of the prediction models. However, the correlation analysis in Section 2.3.2 showed that there are differences in how change type categories correlate with the number of bugs in source files (see columns on the right-hand side of Table 2.5). This is interesting in the context of the relative frequencies of the different categories listed in Table 2.3. For instance, on one the hand *stmt* accounts for more than 70% of all fine-grained changes in our dataset while *func* represents less than 10%. On the other hand, both categories did not show significant differences regarding their correlations (median Spearman rank correlation of 0.63 and 0.6). Hence, the few functionality changes correlate almost as strongly with the number of bugs as the mass of small statement changes. In contrast, *cDecl* exhibits the lowest

ratio of changes and consequently the lowest correlation. These differences motivate us to further investigate how particular change type categories are related to the bug-proneness of a file.

Therefore, the goal of **E 3** is to examine the predictive power of change type categories and how they influence the performance of prediction models. For that purpose we employ *attribute selection* techniques. The idea of attribute selection is to select only those attributes for building prediction models that have a high predictive power with respect to a given class. There are several benefits of narrowing down a dataset only to the relevant attributes. For instance, improved performance as there is a smaller data space to process with a given machine learning algorithm, or the resulting models are smaller and therefore easier to understand [HH03] and interpret. We are mainly interested in the latter benefit.

Information Gain (IG) is a well known and fast technique to evaluate and rank attributes based on their predictive power for a given target class [HH03, MGF07]. IG answers the question: *"How much knowledge do we gain about the bug-proneness of files if we know (a priori) the number of functionality changes, declaration changes etc. of files?"* More formally, IG describes the change in the entropy H of a given class C if the value of an attribute A is known. IG is then defined as

$$IG(C, A) = H(C) - H(C|A) \tag{2.2}$$

The entropy $H(C)$ and the conditional entropy $H(C|A)$ are calculated based on information theory

$$H(C) = -\sum_{c \in C} P(C = c) log_2(P(C = c)) \tag{2.3}$$

$$H(C|A) = -\sum_{a \in A} P(A = a) H(C|A = a) \tag{2.4}$$

Table 2.8: Information Gain values of all change type categories measured with respect to the $bugClass$ of a file (the category with the largest IG value per project is printed in bold).

Project	cDecl	oState	func	mDecl	stmt	cond	else
Compare	0.24	0.30	**1.00**	0.54	0.87	0.02	0.00
jFace	0.00	0.24	0.60	0.76	**1.00**	0.20	0.11
JDT Debug	0.13	0.13	0.64	0.40	**1.00**	0.06	0.00
Resource	0.21	0.54	0.78	**1.00**	0.90	0.10	0.00
Runtime	0.00	0.33	0.50	0.48	**1.00**	0.18	0.10
Team Core	0.37	0.68	**1.00**	0.76	0.68	0.05	0.00
CVS Core	0.00	0.45	0.68	0.40	**1.00**	0.30	0.00
Debug Core	0.08	0.45	**1.00**	0.17	0.76	0.00	0.07
jFace Text	0.04	0.06	0.12	0.46	**1.00**	0.56	0.00
Update Core	**1.00**	0.12	0.24	0.46	0.22	0.05	0.00
Debug UI	0.00	0.16	0.56	0.48	**1.00**	0.29	0.22
JDT Debug UI	0.00	0.28	0.41	0.22	**1.00**	0.33	0.19
Help	0.52	0.96	0.93	**1.00**	0.87	0.00	0.62
JDT Core	0.00	0.42	0.89	**1.00**	0.79	0.56	0.42
OSGI	0.00	0.51	0.99	0.74	**1.00**	0.40	0.09
Median	0.04	0.33	0.68	0.48	**1.00**	0.18	0.07

whereas $P(C = c)$ and $P(A = a)$ is the probability that the variables C and A have the concrete values c respectively a.

Table 2.8 lists the IG values of all change type categories for each project with respect to the target concept *bugClass*, *i.e.*, *bug-prone* or *not bug-prone* (see Equation 2.1).[2] Attributes having a large IG value are highly predictive and should be considered for prediction model building. With a median IG value of 1.00 and having the highest value in 8 out of 15 projects *stmt* shows the largest predictive power in our dataset. The second highest median has *func*. *mDecl* ranks as third (median IG of 0.48) and exhibits the largest IG value for 3 projects. The remaining four categories, *i.e.*, *cDecl*, *oState*, *cond*, and *else*, all have median values significantly below 0.5. Furthermore, these categories show for most projects low IG values and in some cases even have values of

[2]IG requires that numeric attributes are discretized. We discretized all change type categories using the method proposed in [FI93].

0. The result of a *Related-Samples Friedman Test* on the IG values of the change type categories shows a similar picture compared to the correlation values listed in Table 2.5: *stmt*, *func*, and *mDecl* form a group that shows significant larger IG values and are highly predictive compared to the other categories.

Based on the IG values we build *univariate logistic regression models* using the top three change type categories, *i.e.*, *stmt*, *func*, and *mDecl*, as independent variable for all projects in our dataset. Table 2.9 shows the median values for AUC, precision, and recall of the prediction models computed over all projects and for each change type category. Analog to **E 1** and **E 2** the performance measures were computed by applying each model to the dataset it was obtained from. All models were validated using 10 fold cross validation. All three categories show considerably good performance with median AUC values close to 0.8. The best performance was achieved by *stmt* (0.81). A *Related-Samples Friedman Test* showed that the observed differences of the AUC value among *stmt*, *func*, and *mDecl* are not significant.

Although these univariate regression models do not reach the same performance as the models in **E 1** and **E 2**, we can build acceptable prediction models by using only one of three change type categories. Figure 2.2 depicts an excerpt of the plot of the logistic regression model of Debug Core using the change type category *func* as independent variable and *not bug-prone* as target class.[3] One can see the typical sigmoidal shape of the logistic regression model that takes values between 0 and 1 on the vertical axis—the probability of a file that it is *not bug-prone*. Moving rightwards on the vertical axis, *e.g.*, adding methods, decreases the probability of being *not bug-prone*. In exchange for the loss in performance we obtain smaller and more compact models that are easier to interpret. Therefore such a model could be used in practice as a "rule of thumb"-guide. For instance, when implementing a new feature it can be difficult to estimate in advance the amount of textual lines that need to be changed; even estimating all fine-grained changes down to statement level is difficult as it would require a detailed idea of the implementation. In

[3]The performance measures of this particular model are: AUC 0.82, precision 0.77, and recall 0.71.

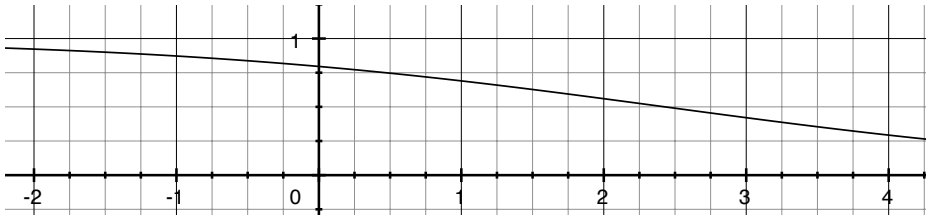


Figure 2.2: Plot of the logistic regression model of Debug Core using func as independent variable and *not bug-prone* as target class.

Table 2.9: Median AUC, precision, and recall values over all projects of the univariate logistic regression model.

Change Category	AUC	Precision	Recall
stmt	0.81	0.73	0.72
func	0.79	0.75	0.71
mDecl	0.77	0.72	0.68

contrast, when using, for instance, an UML class diagram or similar, it might be easier to estimate how many new methods are necessary in a particular file for that feature. Considering the plot given in Figure 2.2, adding three methods decreases the probability of a file for being *not bug-prone* below 50%; adding 4 methods decreases the probability down to 25%—or vise versa increases the probability of a file being *bug-prone* to more than 75%.

In the next step we used *iterative Information Gain subsetting* [MGF07] to find the most predictive subset of change categories for building prediction models using the LibSVM algorithm. We chose LibSVM since it performed the best for more than half of the projects in **E 2** (see Table 2.7). This subsetting procedure starts with the highest ranked category in terms of IG for a given project to build a prediction model. In the next step, the second highest ranked category is included for a new model. We conducted a *Related Samples Friedman Test* after each step. Such a test included the AUC values of all models until the current step and the AUC values of the LibSVM model using all categories (LibSVM column in Table 2.7). This subsetting procedure stops if the pair-wise post-hoc test between the AUC values of the last step and LibSVM exhibits no

Table 2.10: AUC values of step 1–3 of the *iterative Information Gain subsetting* procedure using change type categories as predictor variables and the LibSVM learner.

Project	AUC_{Step1}	AUC_{Step2}	AUC_{Step3}	AUC_{All}
Eclipse Compare	0.78	0.78	0.79	0.81
jFace	0.83	0.88	0.88	0.91
JDT Debug	0.83	0.93	0.94	0.95
Resource	0.84	0.90	0.90	0.92
Runtime	0.80	0.85	0.86	0.87
Team Core	0.78	0.82	0.84	0.86
CVS Core	0.84	0.89	0.89	0.87
Debug Core	0.81	0.89	0.91	0.93
jFace Text	0.77	0.76	0.77	0.79
Update Core	0.81	0.86	0.87	0.89
Debug UI	0.86	0.89	0.91	0.92
JDT Debug UI	0.86	0.88	0.88	0.90
Help	0.64	0.67	0.68	0.69
JDT Core	0.84	0.86	0.87	0.88
OSGI	0.78	0.83	0.86	0.87
Median	0.81	0.86	0.87	0.88

significant differences anymore.

Our data showed that on average including the top three ranked categories of a project results in LibSVM prediction models that perform not significantly different from the ones using all categories in **E 2**. Table 2.10 shows the AUC values obtained based on the models of steps 1, 2 and 3 (columns 2–4) of the subsetting procedure and the AUC values of the LibSVM model that used all change type categories as predictor variables (AUC_{All} column). Using the highest ranked attribute in the first step results in a median AUC value of 0.81. This is equal to the median AUC value of the logistic regression model using *stmt* as independent variable (see Table 2.9). This is not surprising as *stmt* is for more than half of all projects the highest ranked category by means of the IG (see Table 2.8). Including the second ranked category increases the AUC median by 0.05 to 0.86 on average. However, for some projects the increase varies. For instance, Compare does not experiences any increase in its median

AUC at all from step 1 to 2; jFace Text even shows a small decrease from 0.77 to 0.76. The largest AUC improvement is observed in the case of JDT Debug (0.1). The next step—using the top three ranked attributes of each project to train prediction models—increases the median AUC by 0.01 to 0.87. This is close to the value of the LibSVM model that uses all change type categories (median AUC of 0.88) and not significantly different anymore (see also LibSVM column in Table 2.7). In other words, we can build prediction models using only the three highest ranked attributes based on IG that perform equally well compared to models using all categories. Furthermore, this model shows a slightly better median AUC than the models obtained by logistic regression using *LM* as independent variable (median AUC of 0.85) in **E 1** (see Table 2.6).

Summarizing, we further investigated the predictive power of the change type categories using information gain and their relation to the bug-proneness of source files. The results of **E 3** showed that (almost) the same performance can be achieved when using only the top three change type categories. The resulting models showed a better performance than models computed with *LM*. This leads to more compact and practical prediction models. For instance, in case of Update Core the model is based on *cDecl*, *mDecl*, and *func*. These are change type categories that can be anticipated rather early in the development phase compared to *LM* or *SCC*. Therefore, such a model can be applied before the actual implementation and can serve as an indicator of bug-prone classes.

Experiment 4: In the previous experiments 1–3 all prediction models were evaluated using 10 fold cross validation and the classification performance measures were calculated when reapplying a particular prediction model on the *same project* it was trained from. The use of prediction models within a single project assumes the precondition that for each project a substantial quantity of (historical) change and bug data is available to learn such models. However, often not enough data on changes and bug fixes is available for learning prediction models. For instance, the software project is new and only the first release with few data exists.

To address this issue several studies explored the feasibility of cross-project prediction, *i.e.*, applying a prediction model to data different it was trained on.

Those studies included cross-project prediction experiments across different companies, *e.g.*, [ZNG+09, TMBS09], as well as cross-project prediction experiments within the same company, *e.g.*, [BMW02]. Although the mentioned studies differ regarding the used datasets, quantitative methods, and tools the common conclusion that can be distilled is: Cross-project prediction is possible to some extent but does not work in general.

The goal of this experiment is to investigate how *LM* and *SCC* perform for predicting *bug-prone* files across the 15 plugin projects of our dataset. For that we repeated the setup of **E 1** with the exception that performance measures were computed when applying a trained model to all other projects except itself. This resulted in 210 ($= 15 * 14$) cross-prediction pairs, once using logistic regression with *LM* as predictor and once with *SCC* as predictor. Since all plugin sub-projects are part of the Eclipse project, this experiment is of the *"within-company"* nature. According to [ZNG+09], we considered a particular cross-prediction pair as successful if we obtained values $>= 0.75$ for AUC, precision, and recall.

Our results confirm two prior findings. First, cross-projects prediction works only partial and is limited: When using *LM* as independent variable only 3 out of all pairs were successful; using *SCC* substantially increased the number of successful cross-prediction pairs to 24—despite this improvement only about 10% of all cross- prediction experiments were successful in our study. The low recall values are the main reason for the weak performance. Second, the directional nature of cross-project prediction [ZNG+09] is confirmed by our results. The increase in the number of successful cross-prediction pairs when using *SCC* as predictor is mainly due to the fact that Resource and JDT Core could be successfully predicted by more than half of the other projects (9 and 10 projects respectively). In return, neither of both projects could successfully predict any other project.

Table 2.11 shows the median and variance of AUC, precision, and recall calculated over all 210 cross-prediction pairs; once using logistic regression with *LM* and once with *SCC* as predictor variable. With a median AUC of 0.90 SCC achieves significantly (*Wilcoxon Signed Rank Test* with $\alpha = 0.05$)

Table 2.11: Median and variance of AUC, precision (P), and recall (R) calculated over all 210 cross-prediction pairs.

	AUC_{LM}	AUC_{SCC}	P_{LM}	P_{SCC}	R_{LM}	R_{SCC}
Median	0.85	0.90	0.82	0.88	0.45	0.56
Variance	0.01	0.00	0.02	0.00	0.05	0.03

better results compared to *LM* (median AUC of 0.85). Furthermore, while the precision values for both, *LM* and *SCC*, are good (median precision of 0.82 and 0.88 respectively), the recall values are low. Both median recall values are close to 0.5 indicating that on average the number of true positives returned by the logistic regression model is equal to the number of false negatives—in other words only half of the solution, *i.e.*, identifying all *bug-prone* files correctly, is found.

2.3.5 Predicting Bug-Prone Files using different Thresholds

We used the median number of bugs as cut-point in our previous experiments to compute the prediction models. A file was either labeled as *not bug-prone* (*bug-prone*) if the number of bugs of that particular file was smaller (greater) than the median number of bugs per files in a project (see Equation 2.1). This leads to approximately equally sized bins. Almost 43% of all files are labeled as *bug-prone* in our dataset. Consequently, a developer would need to spend a significant amount of perfective maintenance activities when examining all files predicted as *bug-prone*. The goal of this experiment is to (1) analyze how the performance varies with different cut-points, and (2) to compare the performance of *LM* and *SCC* when different cut-points are used for the binning of files. For that, we used three different cut-points:

$$bugClass = \begin{cases} not\ bug-prone & : \quad Bugs <= p_t \\ bug-prone & : \quad Bugs > p_t \end{cases} \tag{2.5}$$

Table 2.12: AUC, precision (P), and recall (R) using logistic regression with *LM* and *SCC* to classify source files into bug-prone or not bug-prone. Models were trained using the 75%, 90%, and 95% percentile as cut-points for the binning of source files. *indicates significantly higher values obtained by one of the models (either using *LM* or *SCC*).

Percentile	75%						90%						95%					
	AUC		P		R		AUC		P		R		AUC		P		R	
Eclipse Project	*LM*	*SCC*	*LM*	*SCC*	*LM*	*SCC*	*LM*	*SCC*	*LM*	*SCC*	*LM*	*SCC*	*LM*	*SCC*	*LM*	*SCC*	*LM*	*SCC*
Compare	.94	.97	.73	.80	.93	.97	.98	.98	.75	.84	1.00	1.00	.79	.90	.72	.93	1.00	1.00
jFace	.95	.95	.68	.68	.94	.96	.96	.97	.50	.60	.98	1.00	.97	.99	.61	.71	1.00	1.00
JDT Debug	.90	.95	.63	.68	.86	.95	.96	.97	.56	.58	.96	.99	.98	.99	.62	.69	.97	1.00
Resource	.91	.95	.66	.70	.91	.94	.97	.98	.68	.73	.98	.96	.98	.98	.73	.72	1.00	.97
Runtime	.90	.97	.59	.76	.92	.95	.93	.98	.48	.77	1.00	1.00	.94	.99	.68	.79	1.00	1.00
Team Core	.71	.91	.40	.65	.85	.90	.82	.95	.29	.59	.94	.95	.86	.97	.30	.64	1.00	1.00
CVS Core	.86	.95	.65	.71	.87	.95	.95	.97	.62	.69	.98	.95	.98	.98	.73	.80	1.00	1.00
Debug Core	.91	.96	.59	.73	.92	.93	.97	.99	.70	.83	1.00	.97	.98	.99	.85	.85	1.00	1.00
jFace Text	.94	.92	.69	.74	.89	.86	.98	.98	.75	.77	1.00	.98	1.00	1.00	.92	.90	1.00	1.00
Update Core	.82	.87	.52	.52	.85	.91	.92	.96	.33	.52	.98	1.00	.95	.97	.49	.58	1.00	1.00
Debug UI	.92	.97	.57	.69	.89	.94	.96	.98	.51	.62	.97	.96	.98	.99	.49	.52	.99	1.00
JDT Debug UI	.97	.97	.76	.75	.95	.94	.98	.98	.70	.64	.97	.99	.98	.98	.60	.66	1.00	1.00
Help	.94	.88	.59	.56	.95	.82	.97	.93	.56	.64	1.00	.87	.97	.93	.60	.70	1.00	.93
JDT Core	.93	.93	.62	.67	.93	.89	.97	.96	.64	.59	.95	.96	.98	.97	.50	.60	.99	.99
OSGI	.92	.88	.71	.69	.85	.85	.95	.92	.49	.63	.97	.92	.97	.96	.51	.76	1.00	.97
Median	.92	.95*	.63	.69*	.91	.94	.96	.97	.56	.64*	.98	.97	.98	.98	.61	.71*	1.00	1.00
Overall	.91	.93	.55	.61	.86	.89	.95	.96	.43	.46	.95	.95	.98	.98	.32	.37	.97	.95

where p_t denotes the value of the 75%, 90%, or 95% percentile of the distribution of bugs in files of a particular Eclipse project.

We build logistic regression models—according to experiment 1 (**E 1**) in Section 2.3.4—once with *LM* and once with *SCC* as predictor variables for each of the above mentioned percentiles. Again, we validated all models using a 10 fold cross validation, and the performance measures were calculated by applying each model on the dataset, *i.e.*, project, it was obtained from.

Table 2.12 lists the AUC, precision, and recall values of the logistic regression models with *LM* and *SCC* as predictors for each percentile. Overall denotes the model that resulted from merging all files of each project in one large dataset. We used a *Related Samples Wilcoxon Signed-Ranks Test* at $\alpha = 0.05$ to test whether the differences in performance (models computed with *LM* compared to models computed with *SCC* as predictor) are significant. For instance, the AUC values of *SCC* for the 75% percentile are significantly higher than the AUC values of *LM* (median AUC of 0.95 vs. 0.92). In contrast, the

small difference in the median AUC values of 0.01 towards *SCC* for the 90% percentile is not significant—and hence possibly observed due to chance in our dataset. We can see that overall the AUC values remain on a high level across all three percentiles for both, *LM* and *SCC*. These values are significantly higher than those in **E 1** in Section 2.3.4 (see Table 2.6) for which the median was used as cut-point.[4] Differences between the AUC values of *LM* and *SCC* shrink, the smaller the number of samples in the predictive target class becomes: We could observe a difference of 0.05 regarding the AUC values between *LM* and *SCC* in **E 1**, the difference decreases to 0.03 in case of the 75% percentile—still significant after all. For the 90% percentile the difference is almost negligible (0.01) small and not significant anymore. In case of the 95% percentile the AUC values of *LM* and *SCC* are equal. One possible explanation of this phenomena might be the heavily skewed distributions of bugs and changes in source files in our dataset. For instance, the target class of the 95% percentile contains only the "top-5 %" files in terms of bugs. Furthermore, those files also accumulate a very large amount of the total changes. For such files, the advantage of *SCC* to capture changes made to source code more accurately (including the semantics of changes) vanishes in the sheer mass of all changes.

While on the one hand recall values are significantly higher for *LM* and *SCC* over all three percentiles compared to the values we observed in in **E 1**, precision on the other hand is significantly reduced. The highest median precision is obtained for *SCC* in the 95% percentile (0.71). When comparing the recall values of *LM* and *SCC* in Table 2.12 the situation is unclear: For the 75% percentile *SCC* shows better recall values (median of 0.94 vs. 0.91), whereas *LM* performs slightly better in the case of the 90% percentile (median of 0.98 v.s 0.97). Both differences were not significant. Similarly to the AUC values, *LM* and *SCC* have equal median recall values for the 95% percentile. In contrast, *SCC* consistently achieves significantly better precision values than *LM* across all three percentiles. The largest difference of the median precision occurs for the 95% percentile (0.61 vs. 0.71). However, one must note that

[4]The median itself represents the value of the 50% percentile of a distribution.

as argued in the beginning of Section 2.3.4 and, for instance in [LBSP08], the use of precision and recall for comparing different classifiers might not be appropriate, in particular with respect to the different percentiles, as they operate with different prior probabilities, *i.e.*, target class distributions.

2.3.6 Predicting the Number of Bugs

In this section, we investigate **H 3**—*SCC* is a better predictor for the number of bugs in Eclipse source files than *LM*.

The most common technique to solve this kind of prediction problem is linear regression. In its simplest case, the relation between bugs and source code changes is modeled as the best fitting straight line, *i.e.*, a linear relationship is established [DWC04]. In [BEP07], Bernstein *et al.* stated that using the nonlinear MP5 regression is more adequate for this kind of data and yields better results when predicting the number of bugs compared to linear regression.

For nonlinear regression analysis, we first need to determine what type of nonlinear function, such as a polynomial, cubic, or exponential, describes the relationship between the dependent and independent variables. Figure 2.3 shows the scatterplot of the CVS Core project on file level. The plot shape is representative for all the Eclipse projects in our dataset.

One can see that a straight line does indeed not fully capture the characteristic of the relationship as stated in [BEP07]. The curve that fits best exhibits a steep slope in the beginning and then flattens out to some extent as *SCC* moves towards large values. This can be interpreted as: When a file already has been subject to a large amount of changes, each additional change is probably less and less important with respect to an increase in *Bugs*. This is similar to the sigmoidal s-shaped function that underlies the logistic regression we used in Section 2.3.4, and that models a saturation effect in terms of probabilities.[5]

An appropriate model for such data as in Figure 2.3 is the asymptotic model described by the function (see [Nor10]):

[5]Logistic regression itself is a nonlinear regression when the dependent variable is non-numerical, *e.g.*, dichotomous.

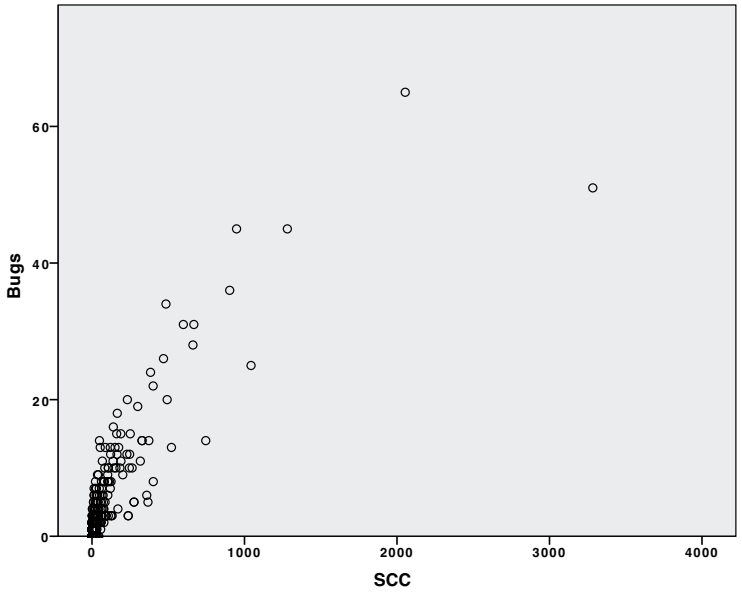


Figure 2.3: Scatterplot between *Bugs* and *SCC* of source files of the Eclipse CVS Core project.

$$f(x) = b_1 + b_2 \times e^{b_3 \times SCC} \; with \; b_1 > 0, b_2 < 0, \; and \; b_3 < 0$$

We used this function to compute the nonlinear regression once with *LM* and once with *SCC* as independent variables and *Bugs* as the dependent variable.

Table 2.13 lists the resulting R^2 values of validating the models with 10 fold cross validation. R^2 is the coefficient of determination that shows how much of the variance in the dataset is explained by the obtained predicting model. *Overall* denotes the performance of the model that resulted when merging all files into one dataset. With a median R^2_{SCC} of 0.79 the models using *SCC* exhibit good explanative power across all projects. Four projects even exhibit an R^2_{SCC} of 0.85 or higher. These models explain a large amount of the variance in their respective dataset. There are three projects in our dataset where nonlinear regression has lower explanative power meaning an

Table 2.13: Results of the nonlinear regression in terms of R^2 and Spearman correlation using *LM* and *SCC* as predictors.

Project	R^2_{LM}	R^2_{SCC}	$Spearman_{LM}$	$Spearman_{SCC}$
Compare	0.84	**0.88**	0.68	**0.76**
jFace	0.74	**0.79**	**0.74**	0.71
JDT Debug	0.69	**0.68**	0.62	**0.8**
Resource	0.81	**0.85**	0.75	**0.86**
Runtime	0.69	**0.72**	0.66	**0.79**
Team Core	0.26	**0.53**	0.15	**0.66**
CVS Core	0.76	**0.83**	0.62	**0.79**
Debug Core	0.88	**0.92**	0.63	**0.78**
Jface Text	0.83	**0.89**	**0.75**	0.74
Update Core	0.41	**0.48**	0.43	**0.62**
Debug UI	0.7	**0.79**	0.56	**0.81**
JDT Debug UI	0.82	0.82	0.8	**0.81**
Help	0.66	**0.67**	0.54	**0.84**
JDT Core	0.69	**0.77**	0.7	**0.74**
OSGI	0.51	**0.8**	0.74	**0.77**
Median	0.7	**0.79**	0.66	**0.77**
Overall	0.65	**0.72**	0.62	**0.74**

$R^2_{SCC} < 0.7$: In Update Core not even half of the variance is explained by the model; in JDT Debug and Help around two third of the variance is explained. An average Spearman correlation of 0.77 indicates the sensitivity of the models, *i.e.*, an accompanied increase/decrease of the actual and the predicted number of bugs.

With an average R^2_{LM} of 0.7, *LM* has less explanatory power compared to *SCC* using an asymptotic model. Except for the case of JDT Debug UI having equal values, *LM* performs lower than *SCC* for all projects including *Overall*. The *Related Samples Wilcoxon Signed-Ranks Test* on the R^2 values of *LM* and *SCC* in Table 2.13 was significant, denoting that the observed differences in our dataset are significant.

To asses the validity of a regression model one must pay attention to the distribution of the error terms. Figure 2.4 shows two examples of fit plots with

normalized residuals (y-axis) and predicted values (x-axis) of our dataset: The plot of the regression model of the *Overall* dataset on the left side and the one of Debug Core having the highest R^2_{SCC} value on the right side. On the left side, one can spot a *funnel* which is one of the "archetypes" of residual plots and indicates that the constance-variance assumption may be violated, *i.e.*, the variability of the residuals is larger for larger predicted values of *SCC* [Lar06]. This is an example of a model that shows an adequate performance, *i.e.*, R^2_{SCC} of 0.72, but where the validity is questionable. On the right side, there is a first sign of the funnel pattern but it is not as evident as on the left side. The lower part of Figure 2.4 shows the corresponding histogram charts of the residuals. They are normally distributed with a mean of 0.

Therefore, we accept **H 3**—*SCC* (using asymptotic, nonlinear regression models) achieves better performance when predicting the number of bugs within files than *LM*. However one must be careful to investigate wether the models violate the assumptions of the general regression model. We analyzed all residual plots of our dataset and found that the constance-variance assumption may be generally problematic, in particular when analyzing software measures and open source systems that show highly skewed distributions. The other two assumptions concerning the error terms, *i.e.*, *zero mean* and *independence*, are not violated. When using regression strictly for descriptive and prediction purposes only, as it is the case for our experiments, these assumptions are less important, since the regression will still result in an unbiased estimate between the dependent and independent variable [Lar06]. However, when inference based on the obtained regression models is made, *e.g.*, conclusions about the slope (β coefficients) or the significance of the entire model itself, the assumptions must be verified.

2.3.7 Analyzing the Effect Size

Throughout this study we analyzed the performance differences by means of *statistical significance testing* (SST). For instance, **Experiment 1** in Section 2.3.4 showed that a logistic regression model benefits from using *SCC* as indepen-

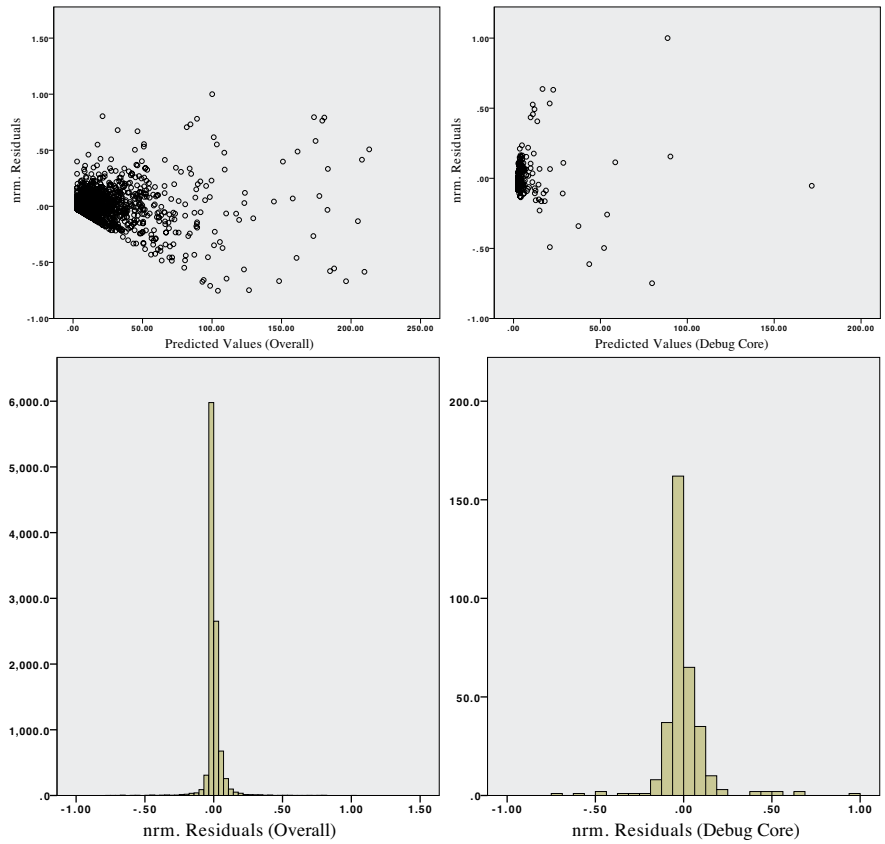


Figure 2.4: Fit plots of the *Overall* dataset (left) and Debug Core (right) with normalized residuals on the y-axis and the predicted values on the x-axis. Below are the corresponding histograms of the residuals.

dent variable; resulting in a *significant* increase of the median AUC from 0.85 to 0.90 (see Table 2.6). However, a drawback of SST is that in some studies a difference of the same size is significant while in other studies not. This fundamental problem arises because statistical significance dependents on two factors: The size of the effect itself, *i.e.*, in our case the performance difference of prediction models when using *SCC* compared to *LM*, and the number of samples. Even an arbitrarily small difference can be still significant assuming that the underlying sample size is large enough [Ker79, DWC04]. Hence, the value of SST for the discussion of the magnitude of an effect is limited. The

basic message of a statistically significant result is: *"Given your data the observed difference is most likely not due to chance."* This makes the use of SST to describe findings of quantitative work across different studies insufficient. It is especially relevant for the field of software engineering where experiments differ widely in many aspects and identical replications are difficult [JV09]. A second problem of significance is that it does not provide any indications and insights for a given difference regarding its (practical) importance [Car93]. In the same manner, as significance does not imply causality, it does not imply importance of a result either [Sch93]. However, researchers often misleadingly include significance as a surrogate for importance and relevance in their results [CHV09]. Therefore, it is recommended that also a discussion about the size of an effect in addition to SST is provided [WT99, Mil05, Mil00, DKS06, KDHS07].

A frequently used effect size metric based on the mean comparison of two groups is *Cohen's d* [Coh88]. d is calculated based on the following equation:

$$\frac{M_1 - M_2}{\sigma_{pooled}} \tag{2.6}$$

where M_1 is the mean of the first group, and M_2 mean of the second group respectively. σ_{pooled} is the pooled standard deviation for the two groups following [Coh88]:

$$SD_{pooled} = \sqrt{\frac{\sigma_1^2 + \sigma_2^2}{2}} \tag{2.7}$$

where σ_1 is the standard deviation of the first group, and σ_2 of the second group respectively. d measures the difference between the means of two groups normalized by their common variance. In other words, d indicates how much the distributions of a measure in two groups overlap. There are mainly two advantages of *Cohen's d*: First, it does not depend on the sample size. Second, its scale free, hence any linear transformation of the measured

Table 2.14: Results of the effect Size calculation using *Cohen's d*.

Experiment	**E 1** (Section 2.3.4)		**E 4** (Section 2.3.4)		75%, 90%, 95% Thresholds (Section 2.3.5)						Non linear regression (Section 2.3.6)	
Table	Table 2.6		Table 2.11		Table 2.12						Table 2.13	
	LM	*SCC*	*LM*	*SCC*	LM_{75}	SCC_{75}	LM_{90}	SCC_{90}	LM_{95}	SCC_{95}	*LM*	*SCC*
$\overline{x}$ AUC, R^2	0.83	0.88	0.83	0.90	0.90	0.94	0.95	0.96	0.97	0.98	0.68	0.76
σ AUC, R^2	0.069	0.058	0.070	0.059	0.062	0.034	0.039	0.019	0.033	0.017	0.166	0.123
Cohen's d	0.78		1.08		0.80		0.33		0.38		0.54	
Cohen's Convention	medium		large		large		small		small		medium	

data does not affect the outcome of d [Hed08]. Therefore, d belongs to the family of standardized effect size measures [KDHS07].[6]

We calculated *Cohen's d* for all prediction experiments in this paper that involved a comparison of the classification performance between *LM* and *SCC*. The first row of Table 2.14 lists the experiment and its corresponding section[7] as well as the original tables containing the performance values of *LM* and *SCC* for which d was calculated: AUC for classification experiments and R^2 for the non-linear regression models. The second row shows the mean ($\overline{x}$), the standard deviation (σ) of the performance values (see Equation 2.6 and Equation 2.7) for both, *LM* and *SCC*, and *Cohen's d*. The last row gives Cohen's own standard convention to describe the magnitude of an effect [Coh88]: $d = 0.2$ discribes a '*small effect*', $d = 0.5$ a '*medium effect*', and $d >= 0.8$ a 'large effect'. With $d = 1.08$ the cross-project prediction models in **E 4** achieve the highest effect when using *SCC* instead of *LM*. This is because the mean difference in **E 4** is the largest among all experiments, *i.e.*, 0.07, but comes with an comparably low σ. Approximately the same d values (0.78 and 0.80) are obtained for the logistic regression model using the median as binning cut-point in **E 1** and the logistic regression model using the 75% percentile as binning cut-point in Section 2.3.5. The logistic regression models based on the 90% and 95% percentile as binning cut-point show the lowest d values (0.33

[6] An introduction into effect size measures as well as a survey of effect size in software engineering experiments can be found in [KDHS07].

[7] We leave out **E 2** and **E 3** since do not directly involve a comparison of *LM* and *SCC* for bug prediction.

and 0.38). Furthermore, for those two cases we did not observe any statistical significant differences between the AUC values of *LM* and *SCC* either (see Section 2.3.5). The last column in Table 2.14—referring to the asymptotic linear regression model discussed in Section 2.3.6—illustrates the basic principle that underlies *Cohen's d* mentioned at the beginning of this section. Although that experiment exhibits the largest mean difference, *i.e.*, 0.08 in terms of R^2, it shows a relatively low d value of 0.54. This comes from the fact that the R^2 values of *LM* and *SCC* have a larger σ compared to the AUC values of the other experiments. Therefore, the distributions of the R^2 values are somewhat "flatter" resulting in more overlap between these two distributions. Hence, a smaller effect size is measured by means of d.

How should one interpret the above listed effect size values in terms of importance? First, effect sizes can be compared to values reported in similar (prediction) experiments or to values in general from the field of software engineering [KDHS07]. In their survey [KDHS07] report a median effect size of 0.60 over all examined articles. Therefore, the observed effect size in **E 1**, **E 4**, and for the 75% percentile are above 0.6 while the remaining effect sizes are moderately below (0.54) and substantially below (0.38 and 0.33). Second, an other possibility of assessing effect sizes is to relate them against a conventional value that is considered to be practical for a given field of interest [KDHS07]. The first possibility requires that a sufficient number of studies report effect sizes in addition to statistical significance. However, this is a delicate task given the fact that effect sizes are rarely discussed in software engineering [KDHS07] as well as in other areas, for instance, educational research [KHL+98]. Furthermore, a common effect size value that is considered to be of practical importance could not be observed by [KDHS07]. In the absence of such benchmarks most work related their effect size to Cohen's original convention (see last row of Table 2.14). However, this convention was made by Cohen based on its observation in behavioral science. Hence, using it in the context of software engineering can be problematic [KDHS07].

Nevertheless, we believe—despite the difficulties of interpreting—discussing the results of statistical tests by means of significance and effect size

is beneficial since it adds another dimension and allows to abstract different studies.

2.3.8 Summary of Results

The results of our empirical study can be summarized as follows:

***SCC* correlates strongly with *Bugs*.** With an average Spearman rank correlation of 0.77, *SCC* has a strong correlation with the number of bugs in our dataset. Statistical tests indicated that the correlation of *SCC* and *Bugs* is *significantly higher* than between *LM* and *Bugs* (accepted **H 1**).

***SCC* categories correlate differently with *Bugs*.** Except for *cDecl* all *SCC* categories defined in Section 2.3.1 correlate substantially with *Bugs*. A *Friedman Test* revealed that the categories have significantly different correlations. Post-hoc comparisons confirmed that the difference is mainly because of two groups of change type categories: (1) *stmt*, *func*, and *mDecl*, and (2) *else*, *cond*, *oState*, and *cDecl*. Within these groups the post-hoc tests were not significant.

***SCC* is a strong predictor for classifying source files into *bug-prone* and *not bug-prone*.** Models built with logistic regression and *SCC* as predictor rank *bug-prone* files higher than *not bug-prone* with an average probability of 90%. They have a significant better performance in terms of AUC than logistic regression models built with *LM* as a predictor (accepted **H 2**).

LibSVM yielded the best performance. In a series of experiments with different classifiers using *SCC* categories as independent variables, LibSVM yielded the best performance—it was the best classifier for more than half of the projects. LibSVM was closely followed by BNet, RFor, NBayes, and NN. Decision tree learners resulted in a significantly lower performance. Furthermore, using all change categories rather than the total number of *SCC* did not yield better performance of prediction models.

Iterative Information Gain subsetting led to more compact prediction models. Using only a subset of all 7 change type categories as input variables, we obtained prediction models with similar performance than the models computed with all changes. For instance, in the case of JDT Core a support vector

machine model based on *mDecl*, *func*, and *stmt* as input variables resulted in nearly the same AUC value as when using all categories (0.87 vs. 0.88).

SCC improves the performance of cross-project prediction. Using a logistic regression model and *LM* as input variable only 1% of all cross-project prediction runs were successful. Using *SCC* instead substantially increased this number to 10%. Additionally, with an median AUC of 0.90 over all cross-project prediction runs *SCC* achieved significantly better performance than *LM*. However, the results showed that cross-project prediction suffers from considerably low recall values.

The better performance of *SCC* to predict *bug-prone* files diminishes as the size of the target class (*i.e.*, bug-prone) becomes smaller. Classification experiments with different cut-points for a priori binning of source files showed that the better performance of logistic regression models using *SCC* diminishes with an increasing cut-point. In case of the 95% percentile, *LM* and *SCC* prediction models achieved the same median AUC value of 0.98.

***SCC* is a strong predictor for the number of bugs in source files**. Asymptotic nonlinear regression using *SCC* showed high explanative power with a median R^2_{SCC} of 0.79 and significantly outperforms the regression models computed with *LM* (accepted **H 3**).

2.4 Discussion and Implications of Results

The results of our study showed that the use of *SCC* improves bug prediction models significantly. The models computed with *SCC* outperformed the models computed with *LM* (*i.e.*, code churn). As a result, our models based on *SCC* can help allocating resources more efficiently to bug-prone parts of a system, *i.e.*, those parts where most of the defects are expected. The gain in performance comes with the additional effort that is needed to extract the fine-grained source code changes from the project history. This is, however, not

an issue when tools, such as CHANGEDISTILLER, are available that perform this extraction fully automatically (*e.g.*, during nightly builds).

The comparison of different classifiers confirmed the results of prior work and showed the strength of advanced machine learning techniques, in particular LibSVM. The importance of the differences in performance should not be overestimated. For instance, the differences between LibSVM, BNet, RFor, NBayes, and NN were not significant in terms of AUC; only the decision tree methods J48 and eCHAID performed significantly lower.

Therefore, to achieve high prediction performance *SCC* should be preferred over *LM* as predictor, and an advanced machine learning technique should be considered for model building. However, the performance difference becomes smaller when a larger percentile cut-point for a priori binning is used, *e.g.*, median vs. 75% percentile (see Section 2.3.5).

Despite the fact that some classification techniques outperform others, an analysis of the distribution of the values and a correlation analysis need to be performed first for this type of experiments. With a value of 0.48 the Help project showed the lowest correlation between *Bugs* and *SCC* (see Table 2.5). Consequently, all selected classifiers resulted in AUC values below 0.7 in that particular case (see Table 2.7). Similar results were obtained for Team Core, which showed almost no correlation (0.15) between *LM* and *Bugs* and consequently low values for AUC (see Table 2.6). This confirms and strengthen the results from prior work, *e.g.*, [NBZ06, ZPZ07]. We recommend an initial correlation analysis before building full-fledged prediction models as it can not only reveal multicollinearity in the dataset but also give a first idea of the strength of the relationship between variables and what their predictive power is.

An analysis of each change type category by means of *Information Gain* showed that—similarly to the correlation analysis—those categories exhibit different predictive power with respect to the bug-proneness of a source file. Selecting only the most predictive categories as input variables leads to more compact and more readable prediction models without sacrificing significantly classification performance. For instance, instead of all categories

using only *mDecl*, *func*, and *oState* to build a prediction model for Team Core results in a model with similar performance. This is of practical interest since the number of required changes for these categories can be estimated roughly already at design time, *e.g.*, using UML class diagrams. In contrast, the number of *LM* and *SCC* can only be measured when parts of the system have been implemented and changes in the versioning system exist. Such a model allows for an earlier identification of *bug-prone* files, and hence for an optimal allocation of testing resources to the most *bug-prone* files (before starting the actual implementation). Moreover, by building prediction models using change type categories managers can prioritize among different types of testing, *e.g.*, integration tests for declaration changes, branch testing when changing conditional expressions, or localized unit tests for statement changes.

The explicit quantification of the empirical relation between a particular category and the bug-proneness of a file enables systematic refactorings to prevent specifically, for instance, declarations changes in the future rather than (textual) changes in general.

In Section 2.3.6, we performed a study to predict the number of bugs in files using regression analysis with *SCC*. The distributions of our dataset and the highly skewed and non-normal distribution of software properties [BFN+06] suggested that linear regression is not appropriate for such data. We recommend to use *nonlinear* regression that better represents the data. The experiments showed that an asymptotic model with a median R^2_{SCC} of 0.79 has high explanative power. The prediction models of 7 projects showed an R^2_{SCC} of 0.8 or higher, 4 had values of 0.85 or higher. An analysis of the residuals indicates that the constance-variance assumption is violated in some cases. Therefore, such models must not be used for inference purposes because the results of (inference) tests are possibly biased. Since we use the models mainly in a descriptive manner, this assumption is of less concern.

2.5 Threats to Validity

From an *external validity* point of view this work is possibly biased by our sole focus on Eclipse projects. Although we collected data from 15 different projects that vary in terms of size, source code changes, and their respective function, they are all part of the larger Eclipse platform. This might question the generalizability of the results and findings for other software systems. In fact, every conclusion based on empirical work is threatened by the bias of the dataset it was drawn from [MGF07]. Especially in software engineering where the development process of a system depends on a large number of factors that potentially vary widely across different systems and domains, the issue of sampling bias may be more prominent [BSL99]. Nevertheless, Eclipse is a representative case study that emerged to a standard IDE since its first release in 2001. It has been studied extensively before, and we can build upon the valuable findings of prior work, *e.g.*, [SZZ06, ZPZ07, MPS08, NAH10]. Therefore, our study contributes to an existing body of knowledge, strengthen existing hypothesis, and presents new results.

Threats to *internal validity* arise from two measurement issues: (1) We counted the number of bugs and established the link between bug data and source files by searching references to bug reports in the log messages of the versioning system (Section 2.2). We rely on the fact that bug fixes are consistently tracked and recorded manually. [BBA+09] reported on evidence about a systematic bias in bug datasets, *i.e.*, the number of bugs appearing in commit messages might not be representative for the corpus of the reported bugs in the bug tracking systems. To address this issue we would need to study whether there is a strong correlation between the number of bugs and the number of reported bugs. Tools, such as Mylyn[8], that seamlessly integrate source code with bug tracking systems could reduce this threat in the near future. (2) When comparing the ASTs of two revisions, CHANGEDISTILLER occasionally extracts a non optimal set of changes, *i.e.*, more changes than

[8] http://www.eclipse.org/mylyn/

actually required for AST transformation. However, the transformation itself between two AST versions is always correct. The accuracy of the change extracting algorithm was evaluated using a benchmark in [FWPG07].

2.6 Related Work

Since software defects are an important cost factor and development teams often operate with limited time and budget constraints, building bug prediction models is an active research field. There are roughly three main factors upon which prediction models are based: Product and process measures and organizational aspects—or a combination of them.

Product measures are directly computed on the source code. In particular, complexity and size metrics have been investigated to build prediction models, *e.g.*, [BBM96, FN99, MGF07]. The rationale is that larger and more complex parts of a system contain more defects. Several approaches used source code dependency information. Findings from [ZN08] showed that the position of a binary within the static dependency graph of Windows Server 2003 correlates with the number of post-release failures. [NAH10] replicated this study on the Eclipse project. In [SZZ06], the import relationship of Eclipse files and packages achieves good predicting power. Similar to our results SVM performed the best. The good predictive power of advanced classifiers, *e.g.*, SVM, Random Forest, and Neural Networks, was confirmed by [LBSP08]. They compared the performance for defect prediction of different learning algorithms using product measures of the NASA dataset. Despite the good performance of some classifiers compared to others, no significant difference could be detected.

Process measures are often obtained from software repositories. *SCC*, as used in this study, falls under this category. Among the first to study the relation between code churn defined as *LM* and *Bugs* was [KAG+96]. Work carried out in [NB05] explored the extent to which the use of relative code churn measures, *e.g.*, *LM* weighted by total lines of code, outperformed absolute measures when

predicting defect density: In Windows Server 2003, absolute churn measures showed a lower performance compared to relative ones. The results of several studies showed that process measures might be better defect predictors than product measures: [GKMS00] found out that the number of changes and the age of a module yield better prediction models than product measures. A comparative study showed that, especially in Eclipse, process measures outperformed product measures [MPS08]. In [KPB06], the J48 decision tree with a combination of product and process measures were used to predict defect density of Mozilla releases. The results showed that process measures are good predictors. The extent to which measuring within different time frames improves bug prediction was investigated in [BEP07]. Consistently with our experiments, prior work validated the usefulness of nonlinear models for building prediction models based on process measures [GKMS00,BEP07]. A study on Windows Vista showed that the number of *consecutive* changes rather than the number of *single* changes have high predictive power [NZZ+10].

Organizational measures describe the management circumstances that influence the development of software. [BND+09] compared the failure differences between components of Windows Vista that were developed in a distributed way and those that were developed at collocated sites . Contrary to common wisdom they stated that geographical differences had little or no effect on failures. A strong organizational indicator of software quality in Windows Vista is developer contribution [PNM08]: The number of developers working on a binary positively correlates with post-release failures. Furthermore, [BNM+11] found out that the structure of the developer contribution network of Windows Vista and Windows 7, *e.g.*, the number of low-expertise developers contribution to particular binaries, are related to failures. Hence, changes made by such *minor contributors* such be reviewed and test with care.

Recent work focused the discussion on prediction models themselves. A critical review about the current state of the art regarding defect prediction models is given in [FN99]. For instance, the authors mention that current prediction models suffer from problems in statistical methods and data quality. Following the results presented in [MGF07] a discussion emerged about

the practical usefulness of defect prediction models [ZZ07, MDDG07]. An overview and comparison of several recent bug prediction approaches can be found in [DLR10a].

Prediction models require a sufficient amount of initial training data. Often, such data is not available beforehand. Therefore, [ZNG+09] raised the importance of exploring the cross-project prediction ability of models, *i.e.*, applying the models to data of a project other than it was obtained from. Their results of reapplying models trained on data from different Microsoft products and several open source projects among each other showed that cross-project prediction is a serious challenge.

2.7 Conclusion and Future Work

In this paper, we empirically analyzed the relationship between fine-grained source code changes (*SCC*) and the number of bugs in source files (*Bugs*) using data from the Eclipse platform. Based on an initial correlation analysis, we computed a set of prediction models using several machine learning methods. The results of our study are:

- *SCC* shows a significantly stronger correlation with the number of bugs than code churn based on lines modified (*LM*) (accepted **H 1**).
- Classification models using *SCC* rank *bug-prone* files higher than *not bug-prone* ones with an average probability of 90%. This is an improvement compared to models computed with *LM* (accepted **H 2**).
- Experiments with different binning cut-points showed that the better classification performance of models computed with *SCC* diminishes as the predictive target class becomes smaller (*i.e.*, cut-point is increased).
- Although advanced learning methods performed better, we could not always observe a significant difference between them.
- Using *Information Gain* to a asses the predictive power of change type categories led to more compact prediction models.

- *SCC* improved the number of successful pair-wise cross-project prediction runs to 10% (out of 210 runs) compared to 1% obtained with *LM*.
- Non-linear asymptotic regression using *SCC* obtained models to predict the number of bugs with a median R^2 of 0.79 which is an improvement over models computed with *LM* (accepted **H 3**).
- The effect size analysis with *Cohen's d* showed that largest effect (d of 1.08) was obtained in case of the cross-project prediction experiment (see Table 2.14).

Our results clearly show the good performance of *SCC* and the improvements over *LM* for bug prediction. This can help allocating maintenance and testing resources to bug-prone parts of a software system.

Currently, our dataset is solely Eclipse focused. Therefore, conclusions made in this work can be biased by characteristics of the development process that are specific and unique to Eclipse. To address this issue replications of our study with other projects are required [BSL99]. Regarding our prediction models, we plan to use different timeframes, *e.g.*, quarterly, half year, or release based timeframes, to investigate how this affects their performance, and how the relation between fine-grained changes and bugs evolves over time. Furthermore, we plan to investigate the relationship between bugs and categories of change types more in depth, *e.g.*, by investigating which change types are used to fix bugs.

The choice of an asymptotic regression model was based on the analysis of the scatterplots. However, more complex or segmented regression models exist that we plan to explore.

3

Method-Level Bug Prediction

Method-Level Bug Prediction
E. Giger, M. D'Ambros, M. Pinzger, H. Gall
Submitted for Conference Publication (Under Review)

Abstract

RESEARCHERS **proposed a wide range of approaches to build effective bug prediction models that take into account multiple aspects of the software development process. Such models have achieved good prediction performance, guiding developers towards those parts of their system where a large share of bugs can be expected. However, most of those approaches predict bugs on file-level. This often leaves developers with a considerable amount of effort to examine all methods of a file until a bug is located. This particular problem is reinforced by the fact that large files are typically predicted as the most bug-prone. In this paper, we present bug prediction models at the level of individual methods rather than at file-level. This increases the granularity of the prediction and thus reduces manual inspection efforts for developers. The models are**

based on change metrics and source code metrics that are typically used in bug prediction. Our experiments—performed on 21 Java open-source (sub-)systems—show that our prediction models reach a precision and recall of 84% and 88%, respectively. Furthermore, the results indicate that change metrics significantly outperform source code metrics.

3.1 Introduction

In the last decade, researchers have proposed a wide range of bug prediction models based on diverse information, such as source code metrics [BBM96, MGF07, ZPZ07, ZN08, NBZ06, Zha09], historical data (*e.g.*, number of changes, code churn, previous defects) [Has09, NZZ+10, NB05, KZWZ07, GKMS00, GPG11], and developers interaction information (*e.g.*, contribution structure) [PNM08, RD11, LNH+11]. Since most prediction models were evaluated on different systems—and frequently with different performance measures—researchers have also investigated which approaches provide the best and most stable performance across different systems [KPB06, MPS08, SJI+10, DLR11].

While having achieved remarkably good prediction performance, most of these approaches predict bugs at the level of source files (or binaries, modules, Java packages). However, since a file can be arbitrarily large, a developer needs to invest a significant amount of time to examine all methods of a file in order to locate a particular bug. Moreover, considering that larger files are known to be among the most bug-prone [BBM96, GFS05, OWB05], the effort required for code inspection and review is even larger. In addition, Posnett *et al.* recently showed that there is a risk of inferential fallacy when transferring empirical findings from an aggregated level, *e.g.*, prediction models at the package- or file-level, to an dis-aggregated, smaller level, for instance, method-level—in particular when such models are used for inspection [PFD11].

In our dataset, a class has on average 11 methods out of which 4 (~32%) are *bug-prone*, *i.e.*, are affected by at least one bug. Assuming that there is only knowledge that a file is *bug-prone*, but not which particular method contains the bug—as given by a file-level prediction model—a developer needs to inspect all methods one by one until the bug is located. Given the median precision of 0.84 achieved by one of our method-level based prediction models (see Table 3.4), a developer has roughly the same chance of picking a *bug-prone* method by randomly guessing after "eliminating" 6 out of those 11 methods ($4/5 = 0.8$). In other words, one needs to manually reduce the set of possible candidates by more than half of all methods until chance is as good as our

prediction models in terms of retrieving a *bug-prone* method. Therefore, we argue that being able to narrow down the location of bugs to method-level can save manual inspection steps and significantly improve testing effort allocation. This is especially important if the resources for quality assurance are limited.

In this paper, we investigate the following research questions:

RQ1 What is the performance of bug prediction models on method-level using change and source code metrics?

RQ2 Which set of predictors, among change metrics, source code metrics, and their combination, provides the best prediction performance at method-level?

RQ3 How does the prediction performance vary if the number of *bug-prone* methods (*i.e.*, positively labeled samples) decreases?

We investigate our research questions based on the source code and change history of 21 Java open-source (sub-)systems. The results of our study show that we can build prediction models that identify *bug-prone* methods with precision, recall, and AUC of 0.84, 0.88, and 0.95, respectively. Furthermore, our experiments indicate that change metrics significantly outperform source code metrics for method-level bug prediction.

In contrast to previous work [KZWZ07] which has also addressed bug prediction on entity-level, the goal of our models is to predict *bug-prone* methods in advance rather than suggesting further *bug-prone* source code entities that need to be changed in addition to that particular entity in which the bug is fixed. Furthermore, we use different methods and metrics to train the prediction models.

The remainder of the paper is organized as follows: Section 3.2 describes our dataset as well as the set of metrics and the tools to compute them. Section 3.3 presents our prediction models and reports on the results of the experiments. We discuss the potential benefits and applications of our approach in

Section 3.4. We present work related to this paper in Section 3.6 and conclude with possible future work in Section 3.7.

3.2 Data Collection

To conduct our prediction experiments we collected a dataset consisting of code, change, and bug metrics for 21 software (sub-)systems (see Table 3.1). Building models to predict bugs at method- rather than at file-level requires that all metrics are available at the method level. In this section, we present the tools and methods necessary to assemble our dataset.

3.2.1 Dataset

We conducted our study with the source code and change history of the projects listed in Table 3.1: #Classes denotes the number of Java classes when checking out the source code at the end of the timeframe (*Time*) from the trunk of the specified repository path; #Methods denotes the number of methods (including Constructors), and #stmt refers to the number of source code statements. #MH is the number of *methodHistories* (see Table 3.3) and #Bugs denotes the number of bugs within the considered timeframe (*Time*). It is possible that #MH<#Methods since there is a substantial amount of methods that are never changed, *e.g.*, accessor-methods or default constructors.

3.2.2 Code Metrics

Code metrics (*i.e.*, product metrics) are directly computed on the source code itself. In the context of bug prediction the underlying rationale of these metrics is that larger and more complex pieces of code are more bug-prone because they are more difficult to understand and to change [DLR11]. In the literature, two traditional suites of code metrics exist: (1) The CK metrics suite and (2) a set of metrics that are directly calculated at the method level that we named

Table 3.1: Overview of the projects used in this study

Project	Version Control System Path	#Classes	#Methods	#MH	#stmt	#Bugs	Time[M, Y]
Compare	dev.eclipse.org:/cvsroot/eclipse Module: org.eclipse.compare	154	1720	2500	12776	563	May01-Sep10
jFace	dev.eclipse.org:/cvsroot/eclipse Module: org.eclipse.jface	374	4438	4043	23991	1275	Sep02-Sep10
JDT Debug	dev.eclipse.org:/cvsroot/eclipse Module: org.eclipse.jdt.debug	436	4434	4700	23517	900	May01-July10
Resource	dev.eclipse.org:/cvsroot/eclipse Module: org.eclipse.core.resources	270	3186	6167	20837	948	May01-Sep10
Team Core	dev.eclipse.org:/cvsroot/eclipse Module: org.eclipse.team.core	157	1510	1124	7833	288	Nov01-Aug10
Team CVS	dev.eclipse.org:/cvsroot/eclipse Module: org.eclipse.team.cvs.core	184	1830	2551	11826	769	Nov01-Aug10
Debug Core	dev.eclipse.org:/cvsroot/eclipse Module: org.eclipse.debug.core	173	1373	2218	6463	493	May01-Sep10
jFace Text	dev.eclipse.org:/cvsroot/eclipse Module: org.eclipse.jface	322	3029	3724	18821	777	Sep02-Oct10
Update Core	dev.eclipse.org:/cvsroot/eclipse Module: org.eclipse.update.core	262	2151	4185	14873	402	Oct01-Junt10
Debug UI	dev.eclipse.org:/cvsroot/eclipse Module: org.eclipse.debug.ui	770	6525	8065	43760	2761	May01-Oct10
JDT Debug UI	dev.eclipse.org:/cvsroot/eclipse Module: org.eclipse.jdt.debug.ui	390	2586	4231	20289	1822	Nov01-Sep10
Help	dev.eclipse.org:/cvsroot/eclipse Module: org.eclipse.help	112	562	536	3503	198	May01-May10
JDT Core	dev.eclipse.org:/cvsroot/eclipse Module: org.eclipse.jdt.core	1140	17703	43134	172939	4888	Jun01-Sep10
OSGI	dev.eclipse.org:/cvsroot/eclipse Module: org.eclipse.osgi	364	4106	5282	27744	1168	Nov03-Oct10
Azureus 3	azureus.cvs.sourceforge.net:/cvsroot/azureus Module: azureus3	362	3983	5394	40440	518	Dec06-Apr10
Openxava	openxava.cvs.sourceforge.net:/cvsroot/openxava Module: OpenXava	507	5132	4656	27662	331	Feb05-Apr11
Jena2	jena.cvs.sourceforge.net:/cvsroot/jena Module: Jena2	897	8340	7764	33542	704	Dec02-Apr11
Lucene	https://svn.apache.org/repos/asf/lucene/dev/trunk/lucene/src/java	477	3870	1754	23788	377	Mar10-May11
Xerces	http://svn.apache.org/repos/asf/xerces/java/trunk/src	693	8189	6866	56920	1017	Nov99-Apr11
Derby Engine	https://svn.apache.org/repos/asf/db/derby/code/trunk/java/engine	1394	18693	9507	116449	1663	Aug05-Apr11
Ant Core	http://svn.apache.org/repos/asf/ant/core/trunk	827	8698	17993	51738	1900	Jan00-Apr11

SCM. The CK suite, introduced by Chidamber and Kemerer [CK94], consists of six metrics that measure the size and complexity of various aspects of object-oriented source code and are calculated at the class level. It was successfully applied for bug prediction in prior work, *e.g.*, [BBM96, SK03]. This suite can be extended by additional object-oriented metrics, such as number of fields per class (*e.g.*, [ZPZ07]). The *SCM* set of metrics is not limited to object-oriented source code, and includes measures such as lines of code (LOC) or complexity. When applied to files, these metrics are typically averaged, summed up over all methods that belong to a particular file, or the highest value in the file is selected [ZPZ07, ZN08, NAH10, LBSP08].

Since our goal is to build bug prediction models at the method level, we do not use the CK suite as it contains metrics which are not directly applicable to methods, *e.g.*, number of sub-classes. We choose instead the metrics listed in Table 3.2, whose good performance were shown in previous studies [MPS08, ZN08, KMM+10].

To compute the code metrics, we first checked out, for each project, the source code version at the end of the timeframe specified in Table 3.1. Then, using the EVOLIZER framework [GFP09], we built a model of the source code

that we use to compute fanIN, fanOUT, localVar, and parameters. Finally, using UNDERSTAND[1], we calculate the remaining code metrics for each method, *i.e.*, commentToCodeRatio, countPath, complexity, execStmt, and maxNesting.

Instead of *lines of code* (LOC) we use the number of declarative (localVar) and executable (execStmt) source code statements per method. We opted for this choice because LOC measures a textual aspect of source files, which is not suitable when changes at the method level are calculated based on the structure of the abstract syntax tree (see Section 3.2.3). However, our data shows that the number of source code statements ($= localVar + execStmt$) approximately corresponds to the LOC per method. In other words, there is roughly one source code statement per line of code.

Table 3.2: List of source code metrics used for the SCM set

Metric Name	Description (applies to method level)
fanIN	Number of methods that reference a given method
fanOUT	Number of methods referenced by a given method
localVar	Number of local variables in the body of a method
parameters	Number of parameters in the declaration
commentTo-CodeRatio	Ratio of comments to source code (line based)
countPath	Number of possible paths in the body of a method
complexity	McCabe Cyclomatic complexity of a method
execStmt	Number of executable source code statements
maxNesting	Maximum nested depth of all control structures

3.2.3 Change Metrics

Version control systems (VCS), such as CVS, SVN, or GIT, contain data regarding the (source code) change history of a software project. VCSs store a

[1]

log entry for each change providing detailed information about that particular change: The file(s) being affected by the change, a (revision) number to uniquely identify each change in correct temporal order, the name of the developer responsible for the change, a timestamp, and a manually entered commit message. Within current VCSs a file typically constitutes the atomic change unit, and hence, changes are solely recorded at the file level. Furthermore, source code files are handled as text files, ignoring their underlying syntactic and semantic structure.

However, to build prediction models at the method level, it is necessary to track changes at a finer granularity. For this purpose, change measures widely adopted for bug prediction [GKMS00, NB05, PNM08, MPS08, BNM+11], such as *number of revisions* and *lines added/deleted*, are too coarse-grained and lack the semantic of individual code changes.

Fluri *et al.* proposed a tree differencing algorithm to extract *fine-grained source code changes* down to the level of single source code statements [FWPG07]. Their algorithm is based on the idea of comparing two different versions of the abstract syntax tree (AST) of the source code, and consists of the following three sub-steps: First, they match all individual nodes between the two versions of the AST using string and tree similarity measures. This matching is required to determine if a particular node was *inserted*, *deleted*, *updated*, or *moved* between two AST versions. In a second step, the algorithm generates a minimal set of these four basic tree edit operations, transforming one version of the AST into the other. Third, each edit operation for a given node is annotated with the semantic information of the source code entity it represents and is classified as a specific *change type* based on a *taxonomy of code changes* [GFP09]. For instance, the insertion of a node representing an else-part in the AST is classified as *else-part insert* change type.

Combining the set of individual tree edit operations resulting from the AST comparison with the semantic information of each node allows us to track source code changes at the fine-grained level of individual source code statements. Moreover, we know not only which particular source entity was changed, but also the exact location of every change within the AST. For

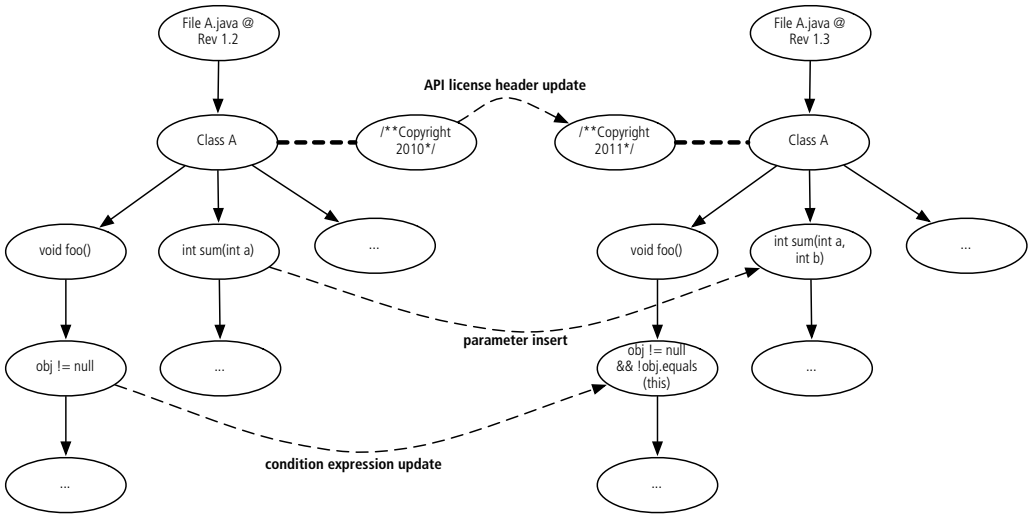


Figure 3.1: A schematic example of the fine-grained code change extraction based on the AST comparison of two file revisions as proposed in [FWPG07].

example, as illustrated in Figure 3.1 it is possible to determine that (1) the condition expression `obj != null` in body of method `foo()` of `Class A` was updated to `obj != null && !obj.equals(this)`, and (2) the parameter `int b` was added to the declaration of method `sum` from revision 1.2 to 1.3 of the corresponding file `A.java`. Furthermore, we are able to distinct between changes that do affect source code entities and "textual" changes, such as license header updates or formatting.

Currently, this tree-differencing algorithm is implemented in CHANGEDISTILLER to work with AST structures of *Java* source code [GFP09]. CHANGEDISTILLER accesses the VCS of a project and pairwise compares all subsequent revisions of every source file. All fine-grained source code changes are then stored in a database. Based on this, we extracted—at the method level—the change metrics (CM) listed in Table 3.3.

We selected and defined these metrics to provide an analogy to file-level based approaches [MPS08]. For instance, *methodHistories* corresponds to the number of revisions of a file; the smt- and churn-metrics in Table 3.3 can be seen as analogue counterparts to the (textual) line based churn metrics. Other

Table 3.3: List of method level CM used in this study

Metric Name	Description (applies to method level)
methodHistories	Number of times a method was changed
authors	Number of distinct authors that changed a method
stmtAdded	Sum of all source code statements added to a method body over all method histories
maxStmtAdded	Maximum number of source code statements added to a method body for all method histories
avgStmtAdded	Average number of source code statements added to a method body per method history
stmtDeleted	Sum of all source code statements deleted from a method body over all method histories
maxStmtDeleted	Maximum number of source code statements deleted from a method body for all method histories
avgStmtDeleted	Average number of source code statements deleted from a method body per method history
churn	Sum of $stmtAdded - stmtDeleted$ over all method histories
maxChurn	Maximum *churn* for all method histories
avgChurn	Average *churn* per method history
decl	Number of method declaration changes over all method histories
cond	Number of condition expression changes in a method body over all revisions
elseAdded	Number of added else-parts in a method body over all revisions
elseDeleted	Number of deleted else-parts from a method body over all revisions

metrics, such as *cond*, are specific to the AST based change extraction.

3.2.4 Bug Data

Bug data of software projects is managed and stored in bug tracking systems, such as Bugzilla. Unfortunately, many bug tracking systems are not inherently linked to VCSs. However, developers fixing a bug often manually enter a reference to that particular bug in the commit message of the corresponding revision, *e.g.*,"fixed bug1234" or "bug#345". Researchers developed pattern matching techniques to detect those references accurately [SZZ05], and thus to link source code files with bugs. We adapted the pattern matching approach to work at method-level: Whenever we find that a method was changed between two revisions of a file (using CHANGEDISTILLER, see Section 3.2.3) and the commit message contains a bug reference, we consider the method to be affected by the bug. Based on this, we then count the number of bugs per method over the given timeframes in Table 3.1.

However, this linking technique requires that developers consistently enter and track bugs within the commit messages of the VCS. Furthermore, we rely on the fact that developers commit regularly when carrying out corrective maintenance, *i.e.*, they only change those methods (between two revisions) related to that particular bug report being referenced in the commit message. We discuss issues regarding the data collection, in particular regarding the bug-linking approach, that might threaten the validity of our findings in Section 3.5.

3.3 Prediction Experiments

We conducted a set of prediction experiments using the dataset presented in Section 3.2 to investigate the feasibility of building prediction models on method-level. We first describe the experimental setup and then report and discuss the results.

3.3.1 Experimental Setup

Prior to model building and classification we labeled each method in our dataset either as *bug-prone* or *not bug-prone* as follows:

$$bugClass = \begin{cases} not\ bug-prone & : \quad \#bugs = 0 \\ bug-prone & : \quad \#bugs >= 1 \end{cases} \tag{3.1}$$

These two classes represent the *binary* target classes for training and validating the prediction models. Using 0 (respectively 1) as cut-point is a common approach applied in many studies covering bug prediction models, *e.g.*, [MPS08, ZPZ07, ZN08, BEP07, MGF07, PNM08]. Other cut-points are applied in literature, for instance, a statistical lower confidence bound [NMB08] or the median [GPG11]. Those varying cut-points as well as the diverse datasets result in different prior probabilities. For instance, in our dataset approximately one third of all methods were labeled as *bug-prone*; Moser *et al.* report on prior probabilities of 23%–32% with respect to *bug-prone* files; in [MGF07] 0.4%–49% of all modules contain bugs; and in [ZPZ07] 50% of all Java packages are bug free. Given this (and the fact that prior probabilities are not consistently reported in literature), the use of precision and recall as classification performance measures across different studies is difficult. Following the advice proposed in [LBSP08, MGF07] we use the *area under the receiver operating characteristic curve* (AUC) to asses and discuss the performance of our prediction models. AUC is a robust measure since it is independent of prior probabilities [BEP07]. Moreover, AUC has a clear statistical interpretation [LBSP08]: When selecting randomly a *bug-prone* and a *not bug-prone* method, AUC represents the probability that a given classifier assigns a higher rank to the *bug-prone* method. We also report on precision (P) and recall (R) in our experiments to allow for comparison with existing work.

In [LBSP08], Lessmann *et al.* compared the performance of several classification algorithms. They found out that more advanced algorithms, such as Random Forest and Support Vector Machine, perform better. However, the

performance differences should not be overestimated, *i.e.*, they are not significant. We observed similar findings in a previous study using fine-grained source code changes to build prediction models on file-level [GPG11]. Menzies *et al.* successfully used Bayesian classifiers for bug prediction [MGF07]. To contribute to that discussion (on method-level) we chose four different classifiers: Random Forest (RndFor), Bayesian Network (BN), Support Vector Machine (SVM), and the J48 decision tree. The Rapidminer Toolkit [MWK+06] was used for running all classification experiments.

We built three different models for each classifier: The first model uses change metrics (CM, see Table 3.3) as predictors, the second uses source code metrics (SCM, see Table 3.2), and the third uses both metric sets (CM&SCM) as predictor variables. All our prediction models were trained and validated using 10-fold cross validation (based on stratified sampling ensuring that the class distribution in the subsets is the same as in the whole dataset).

3.3.2 Prediction Results

Table 3.4 lists the median (over the 10 folds) classification results over all projects per classifier and per model. The cells are interpreted as follows: **Bold** values are significantly different from all other values of the *same performance measure* in the *same row* (*i.e.*, classifier). Grey shaded cells are significantly different from the white cells of the *same performance measure* in the *same row*. To test for significance among the different metric sets we applied a *Related Samples Friedman Test* ($\alpha = 0.05$) for each performance measure (including $\alpha-$adjustment for the pair-wise post-hoc comparison). These tests were repeated for each classifier. For instance, in case of SVM, the median recall value (R) of the combined model (CM&SCM), *i.e.*, 0.96, is significantly higher than the median recall values of the change (0.86) *and* the source code metric model (0.63). With respect to AUC and precision (P), this combined model performed significantly better than the code metric model (AUC: 0.95 vs. 0.7; P: 0.8 vs. 0.48) model but *not* significantly better than the change metric model.

From the performance values one can see two main patterns: First, the

model based on source code metrics performs significantly lower over all prediction runs compared to the change metrics and the combined model. The AUC values of the code metrics model are approximately 0.7 for each classifier—what is defined by Lessman *et al.* as "promising" [LBSP08]. However, the source code metrics suffer from considerably low precision values. The highest median precision value for the code metrics model is obtained in case of J48 (0.56). For the remaining classifiers the values are around 0.5. In other words, using the code metrics half of the methods are correctly classified (the other half being false positives). Moreover, code metrics only achieve moderate median recall values close to 0.6 (except for NB), *i.e.*, only two third of all *bug-prone* methods are retrieved.

Second, the change metrics and the combined model perform almost equally. Moreover, both exhibit good values in case of all three performance measures (refers to RQ1 introduced in Section 3.1). Only the median recall values obtained by SVM and BN for the combined model are significantly higher than the ones of the change metrics model (0.96 vs. 0.86 in both cases). Moreover, while AUC and precision are fairly the same for these two models, recall seems to benefit the most from using both metric sets in combination compared to change metrics only.

Summarizing, we can say that change metrics significantly outperform code metrics when discriminating between *bug-prone* and *not bug-prone* methods (refers to RQ2). A look at the J48 tree models of the combined metrics set supports this fact as the code metrics are added towards the leaves of the tree, whereas except for three projects (~14%) *authors* is selected as root attribute. *methodHistories* is for 11 projects (~52%) the second attribute and in one case the root. Furthermore, considering the average prior probabilities in the dataset (*i.e.*, ~32% of all methods are bug-prone), change metrics perform significantly better than chance. Hence, the results of our study confirms existing observations that historical change measures are good bug predictors, *e.g.*, [GKMS00, MPS08, KMM+10, KPB06]. When using a combined model we might expect slightly better recall values. However, from a strict statistical point of view it is not necessary to collect code measures in addition to change

Table 3.4: Median classification results over all projects per classifier and per model

	CM			SCM			CM&SCM		
	AUC	P	R	AUC	P	R	AUC	P	R
RndFor	.95	.84	.88	.72	.5	.64	.95	.85	.95
SVM	.96	.83	.86	.7	.48	.63	.95	.8	**.96**
BN	.96	.82	.86	.73	.46	.73	.96	.81	**.96**
J48	.95	.84	.82	.69	.56	.58	.91	.83	.89

metrics when predicting *bug-prone* methods.

Regarding the four classifiers, our results are mostly consistent. In particular, the performance differences between the classifiers when based on the change and the combined model are negligible. The largest variance in performance among the classifiers resulted from using the code metrics for model building. However, in this case these results are not conclusive: For instance, on the one hand, BN achieved significantly lower precision (median of 0.46) than the other classifiers. On the other hand, BN showed a significantly higher recall value (median of 0.73).

3.3.3 Prediction with Different Labeling Points

So far we used the absence and presence of bugs to label a method as *not bug-prone* or *bug-prone*, respectively. Approximately one third of all methods are labeled as *bug-prone* in our dataset (see Section 3.3.1). Given this number a developer would need to spend a significant amount of her time for corrective maintenance activities when investigating all methods being predicted as *bug-prone*. Next, we analyze in this section, first, how the classification performance varies generally (refers to RQ3), and second, whether we can observe similar results as in Section 3.3.2 between the change and code metrics model as the number of samples in the target class shrinks (refers to RQ2). For that, we

applied three additional cut-point values as follows:

$$bugClass = \begin{cases} not\ bug - prone & : \quad \#bugs <= p \\ bug - prone & : \quad \#bugs > p \end{cases} \tag{3.2}$$

where p represents either the value of the 75%, 90%, or 95% percentile of the distribution of the number of bugs in methods per project. For example, using the 95% percentile as cut-point for prior binning would mean to predict the "top-five percent" methods in terms of the number of bugs.

To conduct this study we applied the same experimental setup as in Section 3.3.1, except for the differently chosen cut-points. We limited the set of machine learning algorithms to one algorithm as we could not observe any major difference in the previous experiment among them (see Table 3.4). We chose Random Forest (RndFor) for this experiment since its performance lied approximately in the middle of all classifiers.

Table 3.5 shows the median classification results over all projects based on the RndFor classifier per cut-point and per metric set model. The cell coloring has the same interpretation as in Table 3.4: Grey shaded cells are significantly different from the white cells of the *same performance measure* in the *same row* (*i.e.*, percentile). For better readability and comparability, the first row of Table 3.5 (denoted by GT0, *i.e.*, greater than 0, see Equation 3.1) corresponds to the first row of Table 3.4 (*i.e.*, performance vector of RndFor).

We can see that the relative performances between the metric sets behave similarly to what was observed in Section 3.3.2. The change (CM) and the combined (CM&SCM) models outperform the source code metrics (SCM) model significantly across all thresholds and performance measures. The combined model, however, does not achieve a significantly different performance compared to the change model. While the results in Section 3.3.2 showed an increase regarding recall in favor of the combined model, one can notice an improved precision by 0.06 in case of the 90% and the 95% percentile between the change and combined model—although not statistically significant. In

Table 3.5: Median classification results for RndFor over all projects per cut-point and per model

	CM			SCM			CM&SCM		
	AUC	P	R	AUC	P	R	AUC	P	R
GT0	.95	.84	.88	.72	.50	.64	.95	.85	.95
75%	.97	.72	.95	.75	.39	.63	.97	.74	.95
90%	.97	.58	.94	.77	.20	.69	.98	.64	.94
95%	.97	.62	.92	.79	.13	.72	.98	.68	.92

case of the 75% percentile the change and the combined model achieve nearly equal classification performance.

Comparing the classification results across the four cut-points we can see that the AUC values remain fairly constant on a high level for the change metrics and the combined model. Hence, the choice of a different binning cut-point does not affect the AUC values for these models. In contrast, a greater variance of the AUC values is obtained in the case of the classification models based on the code metric set. For instance, the median AUC value when using GT0 for binning (0.72) is significantly lower than the median AUC values of all other percentiles.

Generally, precision decreases as the number of samples in the target class becomes smaller (*i.e.*, the higher the percentile). For instance, the code model exhibits low precision in the case of the 95% percentile (median precision of 0.13). Looking at the change metrics and the combined model the median precision is significantly higher for the GT0 and the 75% percentiles compared to the 90% and the 95% percentiles. Moreover, the median precision of those two percentiles, *e.g.*, 0.64 and 0.68 in case of the combined model, might appear to be low. However, since only 10% and 5% of all methods are labeled as *bug-prone*, this is better than chance.

The picture regarding recall is not conclusive. On the one hand, there are improved median recall values for higher percentiles in case of the code metrics model. For instance, the median recall of the 95% percentile is significantly

higher than the one of GT0 (0.72 vs. 0.64). On the other hand, recall slightly deteriorates for the other two models as higher cut-points for prior binning are chosen. However, one must keep in mind—as stated in Section 3.3.1—that using precision and recall for the comparison of classification models that were obtained under different prior probabilities (*i.e.*, in our case the different percentiles) might not be appropriate.

Summarizing our results, we can say that (even) when the number of samples in the target class diminishes, collecting code metrics in addition to change metrics for building prediction models does not yield better results. Furthermore, the choice of a different cut-point for prior binning does not affect AUC and recall. However, we likely obtain lower precision values.

3.3.4 Summary of Results

Based on the experiments in this section we can answer our research questions posed in Section 3.1.

> *RQ1: It is possible to build method level bug prediction models achieving a precision of 0.85, a recall of 0.95, and an AUC of 0.95.*

Our experiments on 21 different software systems indicate that—using Random Forest—one can build a bug prediction model at the method level which achieves 0.85 precision, 0.95 recall, and 0.95 AUC. Employing different machine learning methods does not significantly impact the performance of the classification, which does not fall below 0.8 for precision, 0.89 for recall, and 0.91 for AUC. This result is similar to the findings of our earlier work performed at the file level [GPG11]. Moreover, in an extensive experiment using 17 different classification algorithms no significant performance differences could be detected [LBSP08]. Hence, instead of using only classification performance as criteria, one might choose an algorithm resulting in a simple

model consisting of a (few) readable rules, such as decision trees.

RQ2: While change metrics (CM) are a stronger indicator of bug-prone methods than source code metrics (SCM), combining CM and SCM does not improve the performance significantly.

CM achieved significantly better prediction results with respect to AUC, precision, and recall (see Table 3.4). For instance, a Random Forest model using CM as input variables obtained a significantly higher median AUC value compared to the same model using SCM as predictor (0.95 vs. 0.72). This confirms prior work: Change metrics outperform measures that are computed from the source code itself [GKMS00, KPB06, MPS08].

While both—the CM based and the combined models—obtain significantly better results than the SCM based model, they are not significantly different among each other. We observed only a slight increase regarding recall when using both metric sets.

RQ3: Choosing a higher percentile for labeling does not affect AUC values.

In addition to the commonly applied criteria "at least one bug" (see Equation 3.1) we used the 75%, 90%, and 95% percentiles (see Equation 3.2) of the number of bugs per methods as cut-point for a priori labeling. We obtained fairly high and consistent AUC values across all four percentiles in case of the CM and the combined models (see Table 3.5). Hence, we conclude that our models are robust with respect to different prior probabilities. Similar observations were made for recall. Not surprisingly, as the number of samples in the target class becomes smaller, *i.e.*, as higher percentiles are chosen as cut-points, precision tends to decrease. Consequently, when comparing prediction models

that were trained with different target class distributions one should use AUC as performance measure as it is independent of prior probabilities [BEP07].

3.4 Application of Results

The results of our study showed that we can build bug prediction models at the method level with good classification performance by leveraging the change information provided by *fine-grained source code changes*. In the following we demonstrate the application and benefit of our prediction model to identify the *bug-prone* methods in a source file compared to a file-level prediction model that performs equally well. For that, we assume a scenario as follows:

A software developer of the JDT Core plugin, the largest Eclipse project, and the Derby Engine module, the largest non-Eclipse project in our dataset, receives the task to improve the unit testing in their software application in order to prevent future post-release bugs. For this, she needs to know the most *bug-prone* methods because they should be tested first and more rigorously than the other methods. For illustration purpose, we assume the developer has only little knowledge about her project (*e.g.*, she is new to the project). To identify the *bug-prone* methods, she uses two prediction models, one model to predict the *bug-prone* source files and our Random Forest (RndFor) model to directly predict the *bug-prone* methods of a given source file.

Furthermore, we take as examples release 3.0 of the JDT Core plugin and release 10.2.2.0 of the Derby Engine module. For both releases, she uses the two prediction models trained on the source code metrics *and* the versioning system history back to the last major release (*i.e.*, 2.1 in case of JDT Core and 10.2.1.6 in case of Derby) for calculating the change metrics. Furthermore, both the models were trained using *1 bug* as binning cut-point (see Equation 3.1) and 10-fold cross validation and then reapplied to the dataset. To better quantify the advantage of our method-level prediction model over the file-level prediction model, we assume that the file-level prediction model performs equally well in terms of AUC, precision, and recall.

Comparison We first discuss two exemplary methods of JDT Core 3.0 in the context of the above outlined scenario and then accordingly two methods of the Derby Engine 10.2.2.0 dataset. We selected these methods because they were ranked and classified as highly *bug-prone* by the RndFor model. Furthermore, they showed a large change history in their datasets.

JDT Core 3.0. On average, 12% of all methods were *bug-prone*, and a class contained, on average, 13 methods in this release of Eclipse. The RndFor model resulted in an AUC of 0.9, precision of 0.82, and recall of 0.93.

In particular, the parent class of the method `Main.configure(..)`[2] had 26 methods in the release revision 1.151 out of which 11 (~42%) were affected by post-release bugs. Our model classified this particular method as *bug-prone* with a probability of 1.0. In fact, (among others) bug `74355`[3] was reported and fixed (rev. 1.162 with 3.1 M2 as target milestone) by changing two conditional expressions. After being guided to the class of this method by her file-level prediction model our developer would have a chance of 42% to guess one of the *bug-prone* methods in the first step. If not successful, her chances increase to 44% (=11/25) in the next step, in the third step to ~46% (11/24) and so on. On the other hand, given the precision of 0.82 achieved by our model[4], she arrives approximately at the same probability of selecting one of the *bug-prone* methods simply by chance after having ruled out 12 methods (*i.e.*, $11/(26-12) = 0.79$ vs. precision of 0.82). Therefore, our model could save up to 12 manual inspection steps.

`LocalDeclaration.resolve(..)`[5] was the only method out of six (as per revision 1.29) that contained a bug. This method was again confidently classified as *bug-prone* by our model with a probability of 0.97. In particular, bug `68998` was reported only a few days after the release and fixed in revision 1.31 for release 3.1. Similarly to the first example, our model correctly identified the affected method and, hence, could prevent a maximum of 5 manual method inspections.

[2] org.eclipse.jdt.internal.compiler.batch.Main.configure(String[])

[3] https://bugs.eclipse.org/bugs/show_bug.cgi?id=<bug_number>

[4] Precision can be seen as the probability that a randomly chosen method is relevant, *i.e.*, contains a bug.

[5] org.eclipse.jdt.internal.compiler.ast.LocalDeclaration.resolve(BlockScope)

Derby Engine 10.2.2.0. The RndFor model created for this release obtained an AUC of 0.9, precision of 0.53, and recall of 0.7. This is a lower performance compared to the model of JDT Core. However, given the fact that only 12% of the methods were *bug-prone* and a class had on average 13 methods, this is better than chance, *i.e.*, predicting bugs at the file level.

The class `CreateIndexConstantAction`[6] had 6 methods as per release revision 429838. `executeConstantAction(Activation)` was the only method being *bug-prone*. Our model correctly classified it with a probability of 0.9. Therefore, more than half of all methods need to be manually "eliminated" until guessing becomes as effective as our model regarding the identification of this particular *bug-prone* method (*i.e.*, $1/(6-4) = 0.5$ vs. precision of 0.53). An analysis of the revisions showed that, for example, bug `2599`[7] was fixed in revision 528033 for the upcoming release 10.3.1.4.

When class `TernaryOperatorNode`[8] was tagged for the release 10.2.2.0 with revision 480219, it contained 30 methods. After this release 6 methods (*i.e.*, 20%) were affected by bugs. One of those methods was `locateBind()`, *e.g.*, bug 2777 was fixed in revision 553735. Again, it was correctly classified as *bug-prone* with a high probability of 0.99. When comparing the prior probability of 20% to the precision of 0.53 our model denotes a major improvement and could save roughly up to 18 manual inspection steps (*i.e.*, $6/(30-18) = 0.5$).

Although these examples show a clear usefulness of our approach, there are some limitations to this scenario as it is illustrated above. For instance, in a corresponding real-life scenario a senior developer is not completely unaware of which particular methods contain most of the bugs. Hence, she will not have to rely on pure guessing when examining the potential candidate methods. Moreover, some methods, *e.g.*, accessor-methods, can be examined rather quickly. However, the scenario clearly shows the benefit of favoring our method-level prediction model over file-level prediction models. Moreover, we are convinced that due to the good performance of our models even senior

[6] org.apache.derby.impl.sql.execute.CreateIndexConstantAction
[7] https://issues.apache.org/jira/browse/DERBY-<bug_number>
[8] org.apache.derby.impl.sql.compile.TernaryOperatorNode

software developers can benefit from them: Our models help to narrow down the search space for identifying the *bug-prone* methods. We plan to investigate these benefits with controlled experiments in future work.

Regarding the practicability of our approach, the overhead of the more complex AST-based structural differencing compared to text differencing, *e.g.*, code churn, is negligible. For instance, the extraction process for the entire Eclipse Compare history takes 5min if the source code revisions are locally available. Currently, the time-critical factor is fetching all source code revisions from a remote repository. Hence, integrating our prediction models into a continuous integration environment, e.g., via svn hook, is part of the future work and would even speed up our approach since the fine-grained source code changes could be calculated locally, *e.g.*, for each commit or during nightly builds.

3.5 Threats to Validity

The Construct Validity of our work, *i.e.*, how accurate we measure a particular concept, is mainly threatened by three facts: First, we establish the link between the change history of a project and bugs by searching for references to bug reports in commit messages. This method is only as reliable as such references are (manually) recorded when committing. In particular, bug reports that are not referenced in commit messages cannot be linked to any revision of the version control system. Therefore, this set of successfully linked bugs might not be a fair representation of all bugs [BBA+09]. We have reduced this threat by taking into account the bug fixing and commit policy as described in the documentation of a particular project. In Lucene, for instance, standard commit patterns are used for bug fixes (e.g. 'lucene-512'), which facilitates the bug-linking.

In addition, this threatens the usefulness of our approach—if bugs cannot be linked we will not be able to train any model. However, analyzing commit messages to establish the link between change history and bug reports is

a common procedure and does also reflect state of the art [SZZ05, DLR11]. Moreover, prior studies found out that bug prediction models are to some extent resistant to such kind of noise [KZWG11]. Recently, research proposed a technique to re-establish links even if they are missing in the commit messages [WZKC11].

Second, CHANGEDISTILLER extracts *fine-grained source code changes* by comparing subsequent file revisions. Hence, varying commit behavior can influence how we measure changes and link bugs. For instance, a developer might commit further changes in addition to a bug fix. In this case we would consider all methods that were changed to be affected by that bug. We mitigated this threat by considering a large number of projects in our experiments. Moreover, in our dataset on average a single method was changed per each revision with a reference to a bug report in its commit message—indicating that bug fixes are regularly committed in isolation. This observation is confirmed by prior studies that in most cases only small changes in a file are committed [PP05]. Moreover, in Eclipse a substantial amount of bugs are indeed fixed in one method [FZG08].

Third, we took all references into account when counting the number of bugs. Therefore, it is possible that not all of these references represent bugs in their sense of meaning [AAP+08], *i.e.*, problems related to corrective maintenance. However, an inspection of bug references referring to JDT Core showed that most of those references are indeed real bugs [DLR11].

The generalizability of our study, *i.e.*, its *External Validity*, is threatened by the dataset we use for this study. For instance, many of the systems belong to the Eclipse ecosystem. Similarly, Derby, Lucene, Ant, Jena, and Xerces are all projects of the Apache Foundation. Therefore, it is possible that our work suffers from the bias opposed by characteristics of the development process unique to these communities. We selected these systems because they are relatively large, actively developed, and were extensively studied before [ZPZ07, GPG11, DLR11, KR11, KMM+10, SJI+10], allowing us to contribute to an existing body of knowledge. In particular, Eclipse emerged to a "de facto standard" case study when analyzing open-source systems. Nevertheless,

all projects are independently developed, come from different domains, and emerged from the context of unrelated communities. Moreover, although open source, Eclipse and (to some extent) Jena have an industrial background.

In addition, all tools used in this paper are publicly available, and Ghezzi and Gall offer our data collecting processes as web services [GG11] facilitating the extension of our work with data from other projects.

We modeled the relation between the two metric sets (see Table 3.2 and 3.3) and bugs in methods using different machine learning algorithms. The quality of our models were discussed by means of their classification performance and statistical significance testing. However, previous literature proposed further metrics, such as past bugs [ZPZ07, KZWZ07], the age of files [MPS08], or developer interaction measures [PNM08, NMB08], as well as different approaches to measure those metrics, *e.g.*, entropy based [Has09], relative [NB05], or burst based [NZZ+10]. As part of our future work we plan to conduct a comparative study with an extended space of metrics including additional attribute selection and data mining techniques.

3.6 Related Work

We discuss related work according to the type of metrics that were used to train the prediction models.

Change Metrics. The idea of change metrics (often referred to as *code churn*) is that bugs are introduced by changes [DLR11]. Thus, the more changes are done to a particular part of the source code the more likely it will contain bugs. In [GKMS00], Generalized Linear Models were built based on several change metrics, *e.g.*, number of changes or average age of the code. A study showed that relative change metrics from the Windows Server change history are better indicators for defect density than absolute values [NB05]. The fault and change history in combination with a (negative binomial) regression model achieved good performance in predicting not only the location, but also the number of bugs [OWB05]. Furthermore, the more complex source code changes are (as

measured by entropy), the more likely they are *bug-prone* [Has09]. Nagappan *et al.* found that the number of subsequent, consecutive changes (rather than the total number of changes) is a strong predictor for bugs [NZZ+10]. Bernstein *et al.* studied the extent to which measuring changes in different timeframes affects prediction performance [BEP07]. In a prior study using the change history of Eclipse, we compared lines based code churn and fine-grained source code changes for bug prediction [GPG11]. The latter metrics resulted in significantly better prediction performance. Shihab *et al.* predicted surprise defects in files that are rarely affected by changes [SMK+11]. An adaptive cache-like approach using fine-grained changes and past-defects to predict bugs at the entity-level (function, method) was proposed in [KZWZ07]. The main difference to our work is that their approach suggests further source code entities that need to be changed while a particular bug is being fixed, rather than predicting *bug-prone* methods in advance.

A study on changes in general showed that a substantial amount of changes are *non-esseantial* changes, *i.e.*, they are not directly related to feature modifying changes [KR11], *e.g.*, adding and removing the keyword `this`.

Code Metrics. Using code metrics for predicting bugs assumes that a more complex piece of code is harder to understand and to change, and therefore, it is likely to contain more bugs [DLR11]. Basili *et al.* investigated the impact of the CK object-oriented metrics suite to software quality [BBM96]. The same metric suite was applied on a commercial system in [SK03]. A set of complexity and size metrics was used to predict post-release bugs in releases of Eclipse [ZPZ07]. The usefulness of (static) code metrics to build prediction models was demonstrated using the NASA dataset [MGF07]. In [LBSP08], an extensive study was conducted with the same dataset, focusing on evaluating different machine learning algorithms. Their conclusion is that the difference between those algorithms is mostly not (statistically) significant. However, this *ceiling effect* is reported to disappear when focusing not only on maximizing detection and minimizing false alarm rates [MMT+10].

The practicability of lines of code (LOC) to predict defects was demonstrated in [Zha09]. El Emam *et al.* showed that the size of a class is a con-

founding factor when building bug prediction models [EBGR01]. An extensive empirical study with 38 different metrics and multivariate models to predict the fault-prone modules of the Apache web-server is presented in [DP02]. Social-network measures were applied on the dependency graph of Windows Server [ZN08] and open-source systems [Has09]: More central binaries are more defect-prone.

Social Measures. Work on this subject investigates how the organizational and social context of the software development process affects its quality. Pinzger *et al.* related social-network techniques to the developer contribution network [PNM08]. They found that if more developers contribute to a certain binary it will more likely be affected by post-release defects. Moreover, removing minor contributors from such a network affects prediction performance negatively [BNM+11]. Recent work showed that investigating code-ownership and interactions between developers at a fine-grained level can substantially contribute to defect prediction [RD11, LNH+11]. Nagappan *et al.* showed that the organizational complexity of the development process is significantly related to defects [NMB08]. Somewhat surprisingly, distributed development does not seem to affect software quality [BND+09].

These metrics are rarely used in isolation but instead are often combined for building bug prediction models [AB06, SJI+10]. The goal is to either achieve (significantly) higher prediction results or to study which of the metrics are better predictors for bugs [DLR11, NMB08]. Although a general consensus has not been achieved, several studies showed—similarly to what we observed in this work—that change metrics potentially outperform code metrics [MPS08, KMM+10, KPB06].

3.7 Conclusions and Future Work

We empirically investigated if bug prediction models at the method level can be successfully created. We used the source code and change history of 21 Java open-source (sub-)systems. Our experiments showed that:

- Change metrics (extracted from the version control system of a project) can be used to train prediction models with good performance. For example, a Random Forest model achieved an AUC of 0.95, precision of 0.84, and a recall of 0.88 (**RQ1**).

- Using change metrics as predictor variables produced prediction models with significantly better results compared to source code metrics. However, including both metrics sets did not improve the classification performance of our models (**RQ2**).

- Different binning values did not affect the AUC values of our models (**RQ3**). Moreover, with a precision of 0.68 our models identify the "top 5%" of all *bug-prone* methods better than chance.

- Conforming prior work, *e.g.*, [LBSP08], we could not observe a significant difference among several machine learning techniques with respect to their classification performance.

Given their good performance, our method-level prediction models can save manual inspection steps.

Currently, we use the entire development history available at the time of data collection to train prediction models. It is part of our future work to measure changes based on different timeframes, *e.g.*, release, quarterly, or yearly based. Furthermore, we plan to investigate a broader feature space, *i.e.*, additional attributes, more advanced attribute selection techniques (rather than "feeding all data" to the data mining algorithms), *e.g.*, Information Gain [MGF07], for prediction model building.

4

Using the Gini Coefficient for Bug Prediction in Eclipse

Using the Gini Coefficient for Bug Prediction in Eclipse
E. Giger, M. Pinzger, H. Gall
Proc. Int'l Workshop on Principles of Softw. Evolution (IWPSE), 2011
DOI: http://doi.acm.org/10.1145/2024445.2024455

THE **Gini coefficient is a prominent measure to quantify the inequality of a distribution. It is often used in the field of economy to describe how goods, *e.g.*, wealth or farmland, are distributed among people. We use the Gini coefficient to measure code ownership by investigating how changes made to source code are distributed among the *developer population*. The results of our study with data from the Eclipse platform show that less bugs can be expected if a large share of all changes are accumulated, *i.e.*, carried out, by relatively few developers.**

4.1 Introduction

Prior work found out that not only properties of the source code itself, *e.g.*, size and complexity, but also the social context of the development process affect the quality of a system. For instance, the number of authors and the contribution structure, *i.e.*, who modified a certain part of the source code, are related to bugs as reported in [PNM08, MPS08, BNM+11].

In this paper we analyze the relationship between code ownership and bugs in source files using the Gini coefficient. This coefficient is well known in the field of economy to measure the disparity of a good's distribution among individuals [Gin12]. Analogously, we apply the Gini coefficient to historical change data and developer information to quantify how source code changes are distributed among developers. In particular we investigate the following two hypotheses:

H 1: The Gini coefficient based on change data correlates negatively with the number of bugs.
H 2: The Gini coefficient based on change data can classify source files into *bug-prone* and *not bug-prone* files.

Our hypotheses are motivated by the rationale that when a few developers contribute a major portion of all changes—resulting in a high Gini coefficient—possibly less bugs occur as there is a clear responsibility and ownership. Whereas the case of the "*too many cooks*-situation" results in more uncoordinated, fragmented, and bug-prone changes.

Furthermore, we examine the extent to which measuring source code changes at three different levels of granularity, *i.e.*, file revisions (*R*), lines modified (*LM*), and fine-grained source code changes (*SCC*) [FWPG07], affects the results of our study.

Our results with data from the Eclipse platform suggest that focusing code

changes on a relatively small group of dedicated developers is beneficial with respect to bugs. In addition, using the Gini coefficient we can compute models to successfully identify *bug-prone* files.

4.2 Gini Coefficient

In this work we understand code ownership by the fact that a relatively small subgroup of developers accumulates a major share of all changes done to a system (or to parts of it). Analyzing this kind of inequality and concentration of a measure is a common task in descriptive statistics when characterizing distributions. Moreover, inequality is often used by economists to describe the disparity of assets in a population. Two examples are: Examining the market concentration by looking at the shares of several companies, and measuring the distribution of wealth among the individuals of a society. In these examples, inequality is usually considered to be unfavorable. Less developed countries typically show a larger inequality in the distribution of wealth among its people. A few big competitors that dominate the market could abuse their power to suppress market mechanisms.

The *Lorenz curve* is a primary graphical method to express inequality and was first used to measure the concentration of wealth [Lor05]. It is a function of the cumulative distribution, *i.e.*, it plots on the x-axis the % of the population against the allocated % of the total wealth on the y-axis. Figure 4.1 shows an example of a Lorenz curve of the Eclipse Resource plugin project. We used developers as the *"population"* and file revision as the *"wealth"* in this figure. A Lorenz curve equal to the diagonal line represents perfect equality where everyone owns the same share of the total wealth. Deviation from the diagonal line means inequality; perfect inequality is where one individual owns everything.

In Figure 4.1 the curve exhibits a serious deviation from the diagonal line, *i.e.*, there is inequality in the distribution of the revisions among the developers that contributed to Eclipse Resource. On the one hand, we can

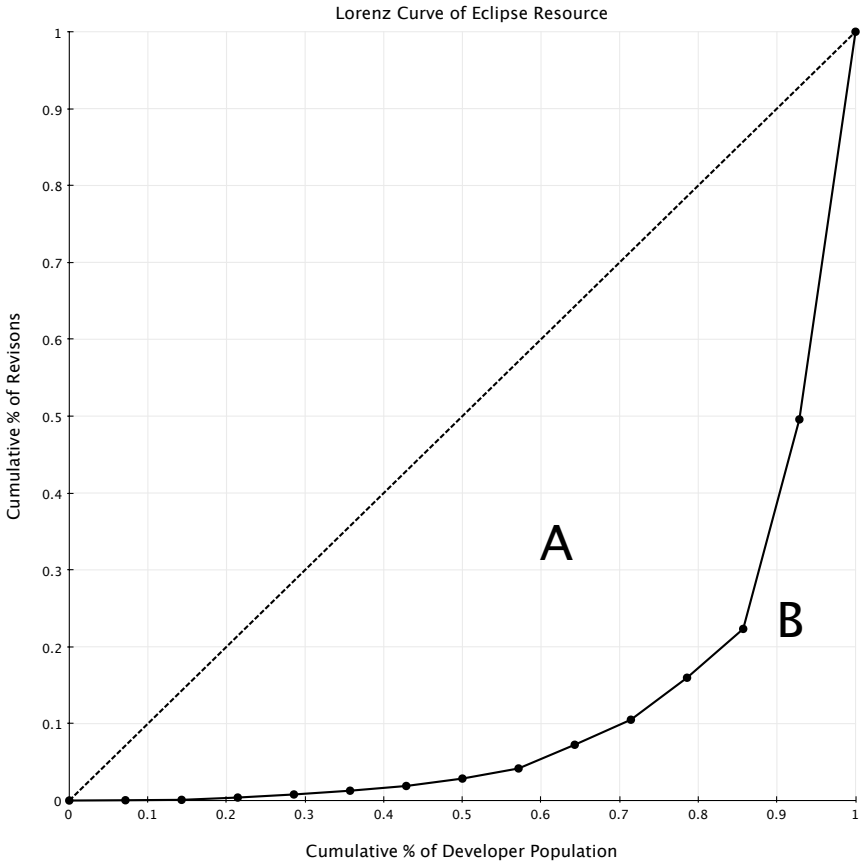


Figure 4.1: Lorenz curve of the Eclipse Resource plugin project using developers and revisions

see that 85% of all developers accumulate only 20% of all revisions. On the other hand, 15% of all developers are responsible for almost 80% of all revisions. Numerically expressed: Eclipse Resource strongly depends on two developers—they committed 6'200 revisions out of 7'932 in total.

The *Gini coefficient* was proposed by Corrado Gini in [Gin12] and is a popular measure of inequality in economy. This coefficient is closely related to the Lorenz curve. In Figure 4.1, A is the area between the diagonal line of perfect equality and the Lorenz curve, B is the area under the lorenz curve. The Gini coefficient is then defined as $A/(A+B)$ [Dor79]. It takes values within the range [0,1]: 0 is perfect equality and 1 reflects perfect inequality. The Gini

coefficient is a robust measure and allows the comparison of the disparity of an attribute of differently sized populations [LBN09] since it does not rely on any assumptions regarding the distribution of that underlying attribute. These characteristics are particularly beneficial in the context of software systems where the distributions of attributes and metrics are often heavily skewed and non-Gaussian [LBN09].

4.3 Data Collection

For the empirical study in Section 4.4 and the computation of the Gini coefficient we collected (1) the historical versioning data, *i.e.*, file revisions (*R*) and lines modified (*LM*) including developer information, (2) fine-grained source code changes (*SCC*), and (3) the number of bugs per file (#*Bugs*).

1. Versioning Data: Versioning repositories, *e.g.*, CVS, GIT, or SVN, provide log entries about the history of a system. Those entries contain information about each revision of all files of that particular system including a manually entered commit message and the name of the developer that committed a revision. *LM* is the sum of lines added, deleted, and changed per each file revision. We use EVOLIZER [GFP09] to automatically access the log entries and extract above mentioned information. Using this versioning information we compute the Gini coefficient for each source file, once based on the distribution of file revisions (Gini_R) and once based on the distribution of *LM* (Gini_{LM}) among developers.

2. Fine-Grained Source Code Changes (SCC): Versioning systems record changes solely on file level and handle source code files internally as text files. Therefore, revisions and *LM* can be too coarse-grained or ignore the semantics of changes to accurately describe all the detailed maintenance activities that occur between two revisions. Fluri *et al.* developed a tree differencing algorithm based on the abstract syntax tree (AST) structure to extract code changes and their semantics at a fine-grained level, *i.e.*, statement level [FWPG07]. The algorithm is part of CHANGEDISTILLER [GFP09] that compares the ASTs of

subsequent file revisions obtained from the versioning system. Including the name of the developer that committed the revision corresponding to a certain version of the AST, we then can compute the Gini coefficient based on the distribution of *SCC* among developers (Gini_{SCC}) for each file.

3. Bugs: Bug repositories, such as Bugzilla, record the information about bug reports of a system. Currently Bugzilla does implicitly not provide direct links to versioning repositories. Therefore, the information which file was affected by a specific bug and when, *i.e.*, in which revision that bug was fixed, is usually missing. However, developers often manually enter a bug report reference, *e.g.*,"*fixed bug 16745*" or "*bug1859*", into the commit messages of file revisions when fixing bugs. Prior research developed matching techniques to query those references and to establish the missing links between bugs and file revisions, *e.g.*, [SZZ05]. Again, we use EVOLIZER to automate this linkage process. Based on this information we then count the number of bugs per file (#*Bugs*).

Gini_R, Gini_{LM} , Gini_{SCC} , and #*Bugs* are then stored in one dataset on file level granularity.

4.4 Study

We investigated our two research hypotheses using data extracted from 16 plug-in projects of the Eclipse platform. Table 4.1 gives an overview of the dataset: *Files* denotes the number of unique `*.java` files. *Rev.* denotes the total number of file revisions. *LM* is the sum of the total number of lines added, deleted, and changed. *SCC* represents to total number of fine-grained source code changes. #*Bugs* is the total number of bugs. *Time* represents the observation period.

Table 4.1: Eclipse dataset used in this study

Eclipse Project	Files	Rev.	*LM*	*SCC*	*#Bugs*	Time [M,Y]
Compare	278	3'736	140'784	21'137	665	May01-Sep10
jFace	541	6'603	321582	25'314	1'591	Sep02-Sep10
JDT Debug	713	8'252	218'982	32'872	1'019	May01-July10
Resource	449	7'932	315'752	33'019	1'156	May01-Sep10
Runtime	391	5'585	243'863	30'554	844	May01-Jun10
Team Core	486	3'783	101'913	8'083	492	Nov01-Aug10
CVS Core	381	6'847	213'401	29'032	901	Nov01-Aug10
Debug Core	336	3'709	85'943	14'079	596	May01-Sep10
jFace Text	430	5'570	116'534	25'397	856	Sep02-Oct10
Update Core	595	8'496	251'434	36'151	532	Oct01-Jun10
Debug UI	1'954	18'862	444'061	81'836	3'120	May01-Oct10
JDT Debug UI	775	8'663	168'598	45'645	2'002	Nov01-Sep10
Help	598	3'658	66'743	12'170	243	May01-May10
JDT Core	1'705	63'038	2'814K	451'483	6'033	Jun01-Sep10
OSGI	748	9'866	335'253	56'238	1'411	Nov03-Oct10
UI Workbench	3'723	38'505	1'427K	168'988	5'000	Sep02-Oct10

4.4.1 Correlation Analysis

In this section we investigate the correlation between the Gini coefficient and the number of bugs on file level (**H 1**). Furthermore, we analyze if there is a difference in the correlation when calculating the Gini coefficient based on *R*, *LM*, and *SCC*. For the correlation analysis we used the Spearman rank correlation ρ. It is more robust than the Pearson correlation since it does not make any assumption regarding the distribution of the data and is not restricted to linear relations between two measured variables [DWC04]. Spearman values of +1 and -1 indicate exceptionally strong correlations, whereas a value of 0 denotes the absence of any correlation. Following [PNM08], we consider correlation values of $-0.5 >= \rho >= 0.5$ as substantial and values of $-0.7 >= \rho >= 0.7$ as strong correlations.

Table 4.2 shows the correlation values of the Gini coefficient based on *R* (Gini_R), *LM* (Gini_{LM}), and *SCC* (Gini_{SCC}) on source file level for each project.

We can see that all correlation values are *negative*. This means that an increase in the inequality of one of the three change measures among all developers—resulting in a larger Gini coefficient—comes with a decrease in the number of bugs. In other words, the more changes in a source file are done by a small group of developers, the less bugs it will have. In contrast, if the changes of a file are scattered more evenly among the developers it is more likely to have bugs.

For all three Gini coefficients the median correlations are below -0.5. With a median of -0.58 Gini_{SCC} has the strongest correlation of all three coefficients and the strongest correlation in 10 out of 16 projects. Furthermore, 10 projects show a substantial correlation, five out of these are even strong correlations. Gini_R has the second highest median of -0.55. Compared to Gini_{SCC} it shows only for three projects the highest correlations. Gini_R has for nine projects substantial correlation values out of which three are strong. CVS Core, Help, and OSGI show no correlation at all. Gini_{LM} exhibits a slightly lower median correlation (-0.54). It has 10 substantial and two strong correlations on project level, and in four cases it shows the highest values.

We used a *Related Samples Friedman Test* to examine these differences between the correlation values of the three Gini coefficients. This test is the non-parametric, rank-based alternative of the *One-Way ANOVA* procedure for comparing related samples. Therefore, we can relax any assumptions regarding the distribution of the data [DWC04]. Since the test was not significant at $\alpha = 0.05$ we conclude that the observed differences of the correlations in Table 4.2 are not significant.

To assess the strength of the correlations, we performed three *One-Sample Wilcoxon Signed Rank Tests* [DWC04] for each Gini coefficient against the hypothesized value of -0.5, *i.e.*, substantial, negative correlation. Similarly to the afore used *Friedman Test*, this test is the non-parametric counter-part of the *One-Sample T-Test*. Again, there are no required assumptions with respect to the distribution of the data. The test only requires a certain degree of symmetry, *i.e.*, approximately the same number of samples above and below the median which is true in our dataset. Furthermore, it can also be applied to smaller

Table 4.2: Non-parametric Spearman rank correlation between *#Bugs* and the Gini coefficients based on *R*, *LM*, and *SCC* on source file level. (* significant at $\alpha = 0.01$)

Eclipse Project	Gini_R	Gini_{LM}	Gini_{SCC}
Compare	-0.68*	-0.69*	**-0.74***
jFace	-0.66*	-0.63*	**-0.71***
Resource	-0.55*	**-0.57***	**-0.57***
Team Core	-0.28*	-0.36*	**-0.4***
CVS Core	-0.05	**-0.5***	-0.4*
Debug Core	-0.35*	-0.34*	**-0.49***
Runtime	-0.33*	-0.42*	**-0.43***
JDT Debug	-0.63*	-0.4*	**-0.7***
jFace Text	**-0.54***	-0.52*	-0.51*
JDT Debug UI	-0.6*	-0.63*	**-0.7***
Update Core	-0.72*	**-0.75***	-0.69*
Debug UI	-0.48*	-0.55*	**-0.59***
Help	-0.08	**-0.34***	-0.29*
JDT Core	**-0.74***	-0.67*	-0.67*
OSGI	-0.09	-0.37*	**-0.47***
UI Workbench	**-0.79***	-0.74*	-0.76*
Median	-0.55	-0.54	**-0.58**

sized samples. All three tests were not significant at $\alpha = 0.05$, *i.e.*, the median correlations of all three Gini coefficients are not significantly different from -0.5. Therefore, we accept **H 1** stated in Section 4.1—*Gini coefficients based on change data correlate (substantially) negatively with the number of bugs*.

4.4.2 Predicting Bug-Prone Files

In the previous Section 4.4.1 we observed a substantial, negative correlation between the Gini coefficient based on the distribution of change data among developers. The purpose of **H 2** is to analyze whether the Gini coefficient of a file can be used to identify *bug-prone* files with reasonable performance. For that we applied six different machine learning algorithms: *Logistic Regression* (LogReg), *Random Forest* (RndFor), and *J48 Decision Tree* (J48) as implemented

by the WEKA toolkit [WF05], *Neural Network* (NN), *Naive Bayes* (NB), and *Support Vector Machine* (LibSVM) as implemented by the RapidMiner toolkit [MWK+06]. The rationale for choosing several learning algorithms stems from previous results presented in [LBSP08] that discovered that *more sophisticated* algorithms, such as NN, LibSVM, and RndFor, or Bayesian Methods [MGF07] achieve possibly better classification performance. We binned all files of each project into the observed, target classes *not bug-prone* and *bug-prone* using the median of the number of bugs per file (#*Bugs*) of that particular project, *i.e.*, equal frequency binning using two classes:

$$bugClass = \begin{cases} not\ bug-prone & : \quad \#Bugs <= median \\ bug-prone & : \quad \#Bugs > median \end{cases} \tag{4.1}$$

We then conducted three different classification experiments each using one of the Gini coefficients as input variable at a time. In each experiment we trained the six learning algorithms on all projects and computed for each project the following performance measures using 10-fold cross-validation: area under the receiver operating characteristic curve statistic (AUC) [MGF07], precision (P), and recall (R).

We mainly use AUC in the performance discussion. AUC is a robust performance measure for classification models since it is independent of prior probability and therefore facilitates the comparison of different approaches [BEP07].

Table 4.3 lists the median of AUC, P, and R over all projects of each learning algorithm for a given Gini coefficient. Except for RndFor and J48, both using Gini_{LM} as input variable, all prediction models obtain AUC values above 0.7, what Lessmann *et al.* denote as *promising results* [LBSP08]. The models computed with the six machine learning algorithms differ in their performance as indicated by the different median AUC values. In the following we discuss these differences with respect to each Gini coefficient. For that, we use the *Related-Samples Friedman Test* using $\alpha = 0.05$. In the case of obtaining a

significant probability, *i.e.*, there is an overall significant difference regarding the AUC values among all learning algorithms, we apply pairwise-post hoc tests including an adjustment of the α-level to investigate between which two learners the differences actually occur:

Gini$_R$. RndFor performs the best with a median AUC of 0.81. All other learners exhibit lower values between 0.73 and 0.75. The *Friedman Test* was barely significant. The post-hoc tests showed that this is due to the better performance of RndFor.

Gini$_{LM}$. NN shows the highest AUC median (0.75). LogReg, NB, and SVM perform only slightly lower than NN. RndFor and J48 have median values below 0.7. The comparable low performance of RndFor was confirmed by the result of the pairwise post-hoc tests as it had the lowest mean rank of all learners. This is surprising as RndFor was the best method in case of Gini$_R$.

Gini$_{SCC}$. NN and LogReg both perform the best with a median AUC of 0.78. With a median AUC of 0.77 NB and LibSVM are second. Similarly to the case of Gini$_{LM}$, RndFor and J48 exhibit the lowest prediction performance. Again, this is confirmed by the *Friedman Test* and the pairwise post-hoc tests. The other algorithms do not exhibit significant differences in terms of their AUC values.

The comparison of the performance of different machine learning methods in our work supports the findings made in [LBSP08]: Some methods might perform better than others but in most cases not significantly. Consequently, the selection of a particular machine learning technique should not be based on classification performance alone.

In Section 4.4.1 we could not observe a significant difference regarding the correlation of Gini$_R$, Gini$_{LM}$, and Gini$_{SCC}$ with *#Bugs*. Analogously, we applied a *Related-Samples Friedman Test* to the AUC values of the best performing learners of each Gini coefficient, *i.e.*, RndFor for Gini$_R$, NN for Gini$_{LM}$, and NN and LogReg for Gini$_{SCC}$. The test was not significant: The Gini coefficients based on the distribution of *R*, *LM*, and *SCC* among developers can equally well discriminate between *not bug-prone* and *bug-prone* files in our dataset. RandFor using Gini$_R$ obtained the highest median AUC

Table 4.3: Median AUC, Precision, and Recall of prediction models computed with each machine learning algorithm (M-Learner) for the three Gini coefficients

	Gini_R			Gini_{LM}			Gini_{SCC}		
M-Learner	AUC	P	R	AUC	P	R	AUC	P	R
NN	0.74	0.65	0.8	0.75	0.66	0.83	0.78	0.7	0.84
LogReg	0.73	0.64	0.78	0.74	0.67	0.81	0.78	0.71	0.82
RndFor	0.81	0.79	0.78	0.69	0.66	0.6	0.72	0.72	0.66
NB	0.74	0.7	0.78	0.74	0.67	0.81	0.77	0.71	0.83
LibSVM	0.73	0.64	0.79	0.74	0.67	0.81	0.77	0.71	0.83
J48	0.75	0.81	0.29	0.65	0.73	0.33	0.74	0.76	0.2

and resulted also in adequate values for precision and recall (0.79 and 0.78 respectively). Therefore, we accept **H 2** stated in Section 4.1—*Gini coefficients based on change data can be used to classify source files into bug- and not bug-prone files.*

4.4.3 Discussion of the Results

In this work we empirically investigated the relation of how changes are distributed among developers and the number of bugs in source files. For that, we computed the Gini coefficient—a prominent economic measure for the inequality of distributions—using three different change measures, *i.e.*, revisions, lines modified, and fine-grained source code changes. The results show that the more changes of a source file are done by a relativly small group of developers, the less likely it will have bugs. The findings suggest that code ownership in terms of changes should be enforced (at least to a certain degree) and contributions made to a file should be focused on a few dedicated developers. Moreover, our models can list potentially *bug-prone* files whose ownership should be re-organized in order to reduce the likelihood of bugs. Since the collection of all required data can be fully automated in a straightforward way and the tools, such as EVOLIZER and CHANGEDISTILLER exist, our models could be integrated into a versioning system for tool support.

We see two main potential threats to the validity our work that need

to be discussed: First, our study used only data from the Eclipse platform. Therefore, our results are possibly biased by the unique characteristics of the Eclipse development process. This fact might threaten the generalizability to systems other than Eclipse. Nevertheless, Eclipse is a mature system that has been subject of numerous studies before, *e.g.*, [MPS08, GPG11]. As such, we can benefit from and contribute to prior knowledge. Furthermore, our results confirm prior findings that the contribution structure of source code, *e.g.*, [PNM08, BNM+11], is related to bugs.

Second, different commit policies and behaviors among the developers can influence the measurement of the change data. For example, some developers might regularly commit (small) individual changes for each modification task, *e.g.*, a single bug fix, while others only commit larger changes, *e.g.*, refactorings including bug fixes. In addition, some projects follow certain commit policies that allow only a set of core developers to commit changes to the versioning repository. The original developers of the changes are not mentioned.

For both threats, additional studies with different systems are required to sustain our findings in this work. Furthermore, it is reported that versioning systems—and hence commit messages—might contain a systematic bias regarding the full population of (reported) bug fixes [BBA+09].

4.5 Related Work

Vasa *et al.* were among the first to describe the distribution of software engineering data using the Gini coefficient [LBN09]. They collected a number of product metrics on class level, *e.g.*, *Number of Methods*, and used the Gini coefficient to analyze how those metrics are distributed among the classes on several systems. Analogously, the Theil index was used to measure the inequality of software metrics in [SvdB10]. Similarly to our work, Winston computed the Gini coefficient based on change data and its distribution among developers [Win08]. However, he used it as a project risk measure rather than an indicator for bugs: The higher the Gini coefficient, the more a project depends

on a few key developers—a situation often termed as *key man risk*. Closer to our work are studies that relate the contribution structure of source code to defects. Pinzger *et al.* found out that the position of Windows Vista binaries in the developer contribution network is an indicator of failures [PNM08]. Bird *et al.* investigated the effect of minor contributors on failures in such networks [BNM+11]. Somewhat contrary, other work stated that the number of developers does not significantly affect bugs [WOB08]. Moser *et al.* presented a comparative study using change and product metrics to predict defects in Eclipse [MPS08]. Among others, their change metrics include the number of authors that committed a file. Schroeter *et al.* showed that import relations of files and packages in Eclipse correlate with their bug-proneness [SZZ06]. In addition to traditional complexity metrics, the structure of the abstract syntax tree of Eclipse source code was used to predict defects [ZPZ07]. Another study on Eclipse investigated the relation between post-release failures and dependency network measures [NAH10].

An extensive review and comparison of more recent bug prediction approaches is given in [DLR10b].

4.6 Conclusions & Future Work

We empirically investigated the relationship between code ownership and bugs in source files. For that we computed the Gini coefficient for each file in our dataset based on the distribution of the changes of that particular file among all developers. We measured changes at three different granularity levels: Revisions, lines modified, and fine-grained source code changes. A high Gini coefficient for a given file means that a relatively small group of developers is responsible for a large amount of changes, *i.e.*, there is a high degree of code ownership for that file with respect to changes. Summarized, the results of our study are:

- The number of bugs in a file correlates negatively with the Gini coefficient: The more changes of a file are done by a few dedicated developers

(high Gini coefficient) the less likely it will have bugs (**H 1**).

- The Gini coefficient on file level can be used to identify *bug-prone* files with adequate performance. The best results (AUC of 0.81) are obtained with a Random Forest prediction model using the Gini coefficient based on the distribution of revisions among developers (**H 2**).

Future work is basically concerned with extending our experiments to systems other than Eclipse. To shed more light on the characteristics of source code changes, we plan to include information about the types of changes. For instance, is it more critical with respect to bugs that code ownership is enforced in case of declaration changes than it is in the case of source code statement changes, *e.g.*, assignments.

5

Can we Predict Types of Code Changes? An Empirical Analysis

Can we Predict the Type of Code Changes? An Empirical Analysis
E. Giger, M. Pinzger, H. Gall
Proc. Int'l Working Conf. on Mining Softw. Repos. (MSR), 2012
accepted for publication

Abstract

THERE exist many approaches that help in pointing developers to change-prone parts of a software system. Although beneficial, they mostly fall short in providing details of these changes. *Fine-grained source code changes* (*SCC*) capture such detailed code changes and their semantics on the statement level. These *SCC* can be condition changes, interface modifications, inserts or deletions of methods and attributes, or other kinds of statement changes. In this paper, we explore prediction models for whether a source file will be affected by a certain type of *SCC*. These predictions are computed on the static source code dependency graph and use social network centrality measures and object-oriented metrics. For

that, we use change data of the Eclipse platform and the Azureus 3 project. The results show that Neural Network models can predict categories of *SCC* types. Furthermore, our models can output a list of the potentially *change-prone* files ranked according to their change-proneness, overall and per change type category.

5.1 Introduction

Researchers have developed methods and tools to better cope with software maintenance and evolution. Some approaches, *e.g.*, [LH93, DJ03, ABF04], use source code metrics to train prediction models, which can guide developers towards the change-prone parts of a software system. The main motivation for these approaches is that developers can better focus on these change-prone parts in order to take appropriate counter measures to minimize the number of future changes [GDL04]. Other approaches, *e.g.*, [ZWDZ04, YMNCC04, RPL08], support developers in modification tasks that affect different source code locations by automatically eliciting past changes and change couplings between these source code entities. Moreover, the sensitivity to which the design of a system reacts to changes can be an indicator for its quality [TCS05].

While the results of existing approaches are promising they fall short in providing insights into the details of changes. In particular, most of the current prediction models are based on coarse-grained change measures, such as code churn (lines added/deleted) or number of file revisions, *e.g.*, [ZL07, ABF04]. These measures, however, do not capture the details about the semantics of changes. For instance, they do not provide detailed information whether a condition expression has changed or the declaration of a method was modified.

We explore in this paper to which extent data-mining models can predict if a source file will be affected by a certain category of source code change types, *e.g.*, declaration changes. For that, we leverage the (semantic) change information of *fine-grained source code changes* (*SCC*) [GFP09]. To compute the prediction models we focuse on object-oriented metrics (*OOM*) [CK94] and centrality measures from social network analysis (*SNA*) [WF94] computed on the static source code dependency graph since they showed explicitly well predicting performance and in some cases achieved better performance than traditional metrics—both for change [LH93] and bug prediction [ZN08].

Being able to predict not only if a file will most likely be affected by changes but additionally by what types of changes has practical benefits. For example, if a developer is made aware that there will be API changes she can plan

accordingly and allocate resources for systemwide integration tests with dependent modules and, furthermore, she might account for additional time to update the API and design documents. In contrast, if only small statement changes are predicted localized unit tests will be sufficient and no further change impact can be expected.

In particular, we formulate two hypotheses:

H 1: *OOM* and *SNA* measures correlate positively with fine-grained source code changes.
H 2: *OOM* and *SNA* measures can predict categories of source code changes.

We investigate these hypotheses by a quantitative and manual analysis of 19 Eclipse plug-in projects as well as the Azureus 3 project. The results of our studies show that *OOM* and *SNA* metrics can be used to compute models to predict the likelihood that a source file will be affected by a certain category of source code changes. For instance, the models for predicting changes in the method declarations of Java classes obtained a median precision of 0.82 and a recall of 0.77. In all our models, the complexity of classes as well as the number of outgoing method invocations show the highest correlation and predictive power for our change type categories.

The remainder of the paper is organized as follows: Section 5.2 describes the process of data collection. Section 5.3 contains the empirical study with respect to our hypotheses and the manual analysis. Section 5.4 provides a discussion of the findings. We describe related work in Section 5.6. Section 5.7 points out possible future work.

5.2 Data Collection

In this section we describe our approach, and the methods and tools we used in this paper. We need four kinds of information to prepare the dataset for our experiments in Section 5.3: (1) Source code dependency graph; (2) Centrality measures from social network analysis [WF94] based on the dependency graph; (3) object-oriented source code metrics [CK94]; (4) and fine-grained source code changes [GFP09].

Dependency Graph. The dependency graph of a software system depicts the relational structure between individual source code entities. We use the EVOLIZER suite [GFP09] to extract the dependency information based on the version of the source code checked out from the trunk at the end of the time-frames listed in Table 5.3 for each project. We consider the following set of dependencies for our study: Method invocation, field access, inheritance, type cast, and instance-of check. We aggregated all dependency information on file level granularity. Hence, the nodes in the graph represent files and the edges indicate the existence of a dependency between two files. Similar to Zimmermann *et al.* [ZN08], we distinguish between in- and outgoing connections and allow for self-connections, *i.e.*, a file can have dependencies to itself. In contrast to [ZN08], however, we include weighted connections defined by the number of dependencies between two files. We choose weighted edges since centrality measures computed from an unweighted dependency graph showed lower correlation with *SCC* in our dataset. The result is a file-based, directed dependency graph in which edges are labeled with the number of dependencies.

Centrality Measures (*SNA*) stem from social network analysis and characterize the concept of *centrality* that identifies nodes in "*favored positions*" with more power [WF94]. Therefore, files having more ties to other files are "*more central*" and can be interpreted as more important. In practice, several approaches exist to measure the concept of centrality, see Table 5.1. Centrality measures computed on the static dependency graph performed explicitly well for (bug) prediction purposes, *e.g.*, [NAH10]. Moreover, in some cases they achieved

better results than traditional metrics, *e.g.*, LOC, [ZN08]. Hence, regarding **H 1** and **H 2**, we hypothesize that files which are more central and have more connections to others files are more *change-prone*.

For an overview of social network analysis we refer to [WF94]. We computed the centrality measures on the afore extracted file dependency graph with UCINET [BEF99]. All measures were obtained at file level.

Object-Oriented Metrics (*OOM*) are a set of well established metrics measuring the size and complexity of object-oriented systems [CK94], see Table 5.2. Prior work demonstrated their usefulness for building prediction models: For defect prediction, *e.g.*, [BBM96], as well as change prediction models, *e.g.*, [LH93, ZL07]. The underlying rationale for our work is that more complex parts of a system are more likely to face changes. Again, the object-oriented metrics were computed on the version of the source code checked out from the trunk at the end of the timeframes listed in Table 5.3 for each project using UNDERSTAND (). We aggregated all metrics on file level.

Fine-Grained Source Code Changes (*SCC*). Version Control Systems (VCS) such as CVS, SVN, or GIT handle source files as pure text files ignoring their implicit code structure. Therefore, change measures such as lines added/deleted are rather imprecise since they can indicate code changes although no source code entities were changed, *e.g.*, in case of text formatting. Furthermore, they can not distinguish between different types of changes; changing the name of a class or the parameter list of a method declaration will both likely result in "+1 line changed". Another problem is that recording changes solely on file level, *i.e.*, revisions, can be too coarse grained: In our dataset around 8 distinct source code entities of the same file were changed per revision. Fluri *et al.* developed a tree differencing algorithm to extract *fine-grained source code changes* (*SCC*) [FWPG07]. They leverage the implicit structure of source code by comparing two different versions of the abstract syntax tree (AST) of a program and can track source code changes down to statement level. The algorithm is implemented in CHANGEDISTILLER [GFP09]. This tool compares the ASTs of each pair of subsequent revisions of all files of a system provided by its VCS. We applied CHANGEDISTILLER to the VCSs of all projects and

Table 5.1: Description of Network centrality measures (*SNA*)

Network Centrality Measures	
Approach	Measure (n=normalized [0-1])
Degree Centrality measures the concept of centrality based on the immediate ties of a file, *i.e.*, the more ties the more central a file is.	**Freeman Degree** counts the number of immediate ties a file has with other files. We distinguish between outgoing (*nOutDegree*) and incoming (*nInDegree*) ties [Fre79].
	Bonacich's Power (*nPower*) computes centrality based on the number of immediate ties of a file, and on the number of immediate ties of its neighbors [Bon87].
Closeness Centrality includes the indirect ties and focuses on the distance of an individual file to all other files in the dependency graph.	**Freeman shortest path closeness** is the reciprocal of the sum of the lengths of all shortest paths from a file, *i.e.*, *outCloseness* (or to a file, *i.e.*, *inCloseness*) to all other files in the graph [Fre79].
	Reachability is the number of files reachable from a file (*nOutReach*) or which can reach a file (*nInReach*) within 1..n hops [BEF99].
Freeman Betweenness (*nBetweeness*) determines how often a file is part of the shortest path between two other files [Fre79].	

extracted all *SCC* that occurred during the timeframes listed in Table 5.3 for each file.

5.3 Empirical Study

This section presents the empirical study we carried out to investigate our hypotheses formulated in Section 5.1. We describe the dataset, the statistical methods we applied, and report on the results and findings.

Table 5.2: Description of the Object-oriented metrics (*OOM*)

Object-Oriented Metrics [CK94]
Weighted methods per class (WMC) is the sum of the cyclomatic complexity of all methods of a class.
Coupling between object classes (CBO) counts the coupling to other classes.
Lack of cohesion in methods (LCOM) counts the number of pairwise methods without any shared instance variables, minus the number of pairwise methods that share at least one instance variable.
Depth of inheritance tree (DIT) denotes the maximum depth of the inheritance tree of a class.
Number of children (NOC) is the number of direct subclasses of a class.
Response for class (RFC) counts the number of local methods (including inherited methods) of a class.

5.3.1 Dataset

We conducted our study with 19 plugin projects of the Eclipse platform and the Azureus 3[1] project. They are well established in their domains, have a maintenance history of several years, and were often subject to prior research, *e.g.*, [SZZ06, ZPZ07, MPS08, NAH10, GPG11].

Table 5.3 gives an overview of our dataset: #Files denotes the number of unique **.java* files we obtained when checking out the source code version at the end of the timeframe (Time) from the trunk of the version control system. #Rev denotes the total number of revisions of the given source files within the timeframe, #LM denotes the total number of lines added and deleted, and #*SCC* is the total number of fine-grained source code changes.

An initial investigation of the dataset revealed large differences in how often certain *SCC* types occurred. In order to have higher frequencies for our experiments we combined several change types into one change type category according to their semantics (see Table 5.4).

Some change types such as *adding attribute modifiability* (removing the

[1], CVS-Path: Module: azureus3

Table 5.3: Dataset used in this study (DB=Debug)

Project	#Files	#Rev	*#LM*	*#SCC*	Time[M, Y]
Compare	154	2'953	111'749	17'263	May01-Sep10
jFace	378	5'809	304'744	22'203	Sep02-Sep10
JDT DB Jdi	144	1'936	63'602	6'121	May01-July10
JDT DB Eval	105	1'610	27'337	6'091	May01-July10
JDT DB Model	98	2'546	78'225	12'566	May01-July10
Resource	274	6'558	260'298	28'948	May01-Sep10
Team Core	169	1'995	38'317	4'607	Nov01-Aug10
CVS Core	188	5'448	157'176	23'301	Nov01-Aug10
DB Core	187	3'033	76'594	12'342	May01-Sep10
jFace Text	312	4'980	107'461	23'633	Sep02-Oct10
Update Core	275	6'379	151'823	27'465	Oct01-Jun10
DB UI	788	10'909	281'485	57'075	May01-Oct10
JDT DB UI	381	5'395	108'920	28'956	Nov01-Sep10
Help	110	999	20'661	5'919	May01-May10
JDT Compiler	322	19'466	1'099K	171'915	Jun01-Sep10
JDT Dom	157	6'608	233'105	32'699	Jun01-Sep10
JDT Model	420	16'892	596'320	90'128	Jun01-Sep10
JDT Search	115	5'475	201'876	44'372	Jun01-Sep10
OSGI	395	6'455	239'430	38'203	Nov03-Oct10
Azureus 3	368	6'327	187'869	46'232	Dec06-Apr10

keyword `final` from an attribute declaration) account—even if combined—for less than one percent of all changes and are left out in our analysis (see [GFP09] for a list of all change types).

5.3.2 Correlation Analysis

We use the Spearman rank correlation to analyze the correlation of *SNA* and *OOM* metrics with *#SCC* (**H 1**). We choose the Spearman over Pearson correlation because it does not make any assumptions about the distribution of the data and measures the strength of a monotonic relation (rather than linear) between two variables [DWC04]. The Spearman correlation obtains values between +1 and -1: +1 represents a high positive and -1 a high negative

Table 5.4: Categories of change types used in this study.

Category	Description
cDecl	Combines all changes that modify the declaration of a class, *e.g.*, changing the class name.
func	Combines the insertion and deletion of functionality, *i.e.*, adding or removing methods.
oState	Combines the insertion and deletion of object states, *i.e.*, adding or removing class attributes.
mDecl	Combines all changes that modify the declaration of a method, *e.g.*, changing the method name.
stmt	Combines all changes that modify executable statements, *e.g.*, changing the name of local variable.
cond	Combines all changes that modify the condition expression in control structures.
else	Combines the insertion and deletion of *else*-parts.

correlation between two variables. We consider values below -0.5 and above +0.5 as substantial correlations [PNM08], and values below -0.7 and above 0.7 as strong correlations [DJ03,PNM08]. We first examine the correlation between #*SCC* and centrality measures (*SNA*) and then analyze the correlation between #*SCC* and object-oriented metrics (*OOM*).

Centrality Measures: Table 5.5 lists the results of the correlation analysis per project. With a median correlation of 0.66 and exhibiting the largest value for 16 projects *nOutDegree* shows the strongest correlation of all centrality measures. For 8 projects we can observe strong correlations; 2 out of them have values of 0.8 and above. Only 3 projects are below 0.5. *nPower* is close to *nOutDegree* with the second highest median (0.65). 8 projects have strong correlations and 4 are below the level 0.5. The third highest median (0.61) has *nInDegree*. With a median of 0.53 *nOutReach* has a substantial correlation with #*SCC* on average.

outCloseness and *nBetweenness* both do not have a substantial correlation on average, but their values above 0.4 indicate an existing positive correlation that might have discriminatory power when building prediction models [ZN08].

Table 5.5: Spearman rank correlation between directed network centrality measures and *#SCC* at the level of source files. * marks significant correlations at $\alpha = 0.01$. The largest value is printed in **bold**.

Project	nOutDegree	nInDegree	nPower	outCloseness	inCloseness	nOutReach	nInReach	nBetweenness
Compare	**0.77***	0.67*	0.74*	0.49 *	-0.2	0.68*	0.06	0.68*
jFace	**0.75***	0.64*	0.74*	0.51*	-0.12	0.59*	0.02	0.42*
JDT DB Jdi	0.72*	**0.77***	0.72*	0.08	0.31*	0.17	0.26*	0.4*
JDT DB Eval	**0.45***	0.41*	0.44*	-0.02	0.12	0.11	0.12	0.33
JDT DB Model	0.6*	**0.71***	0.57*	0.42*	0.07	0.52*	0.44*	0.56*
Resource	**0.68***	0.64*	0.65*	0.45*	-0.08	0.56*	0.35*	0.55*
Team Core	0.45*	**0.50***	0.35*	0.23*	0.21*	0.29*	0.25*	0.35*
CVS Core	**0.76***	0.62*	0.75*	0.26*	0.13	0.64*	0.28*	0.55*
DB Core	0.45*	**0.49***	0.42*	0.21*	0.03	0.37*	0.25*	0.35*
JFace Text	**0.66***	0.64*	0.65*	0.57*	-0.22*	0.58*	-0.18*	0.41*
Update Core	**0.62***	0.60*	0.61*	0.42*	0.02	0.49*	0.13	0.37*
DB UI	**0.65***	0.54*	0.40*	0.21*	0.14*	0.47*	0.19*	0.46*
JDT DB UI	**0.56***	0.52*	**0.56***	0.31*	0.15*	0.34*	0.17*	0.31*
Help	**0.52***	0.45*	0.5*	0.42*	-0.2	0.52*	0.09	0.42*
JDT Compiler	**0.85***	0.63*	**0.85***	0.63*	0.06	0.70*	0.40*	0.63 *
JDT Dom	**0.76***	0.67*	0.75 *	0.51*	0.01	0.58*	0.13	0.36*
JDT Model	**0.66***	0.54*	**0.66***	0.52*	-0.25*	0.53*	0.32*	0.44*
JDT Search	**0.8***	0.71*	0.76*	0.55*	-0.01	0.56*	0.25*	0.65*
OSGI	**0.52***	0.48*	**0.52***	0.34*	-0.09	0.38*	0.18*	0.41*
Azureus 3	**0.71***	0.59*	**0.71**	0.55*	-0.01	0.58*	0.06	0.49*
Median	**0.66**	0.61	0.65	0.42	0.02	0.53	0.19	0.42

With an average of 0.02 and 0.19 *inCloseness* and *nInReach* do not exhibit any correlation and will be excluded from the prediction experiments.

The values in Table 5.5 indicate that there are differences regarding the correlations of certain centrality measures and *#SCC*. To investigate these differences, we first performed the *Related Samples Friedman Test* that was significant at $\alpha = 0.05$. We then used pair-wise post-hoc tests to statistically investigate the differences between individual centrality measures. We adjusted the α-level using the *Bonferroni Procedure* [DWC04] for all post-hoc tests. Although the decisions are borderline, the post-hoc tests confirmed that measures based on outgoing connections, *i.e.*, *nOutDegree*, *nOutReach*, and *outCloseness* have significantly higher correlations than their corresponding measures based on incoming connections, *i.e.*, *nInDegree*, *nInReach*, and *inCloseness*.

Object-Oriented Metrics: Table 5.6 shows the Spearman rank correlation values between object-oriented metrics and *#SCC* for each project. With a median of 0.73 WMC shows the highest correlation of all metrics for each

Table 5.6: Spearman rank correlation between object-oriented metrics and *#SCC* at the level of source files. * marks significant correlations at $\alpha = 0.01$. The largest value is printed in **bold**.

Project	WMC	CBO	LCOM	DIT	NOC	RFC
Compare	**0.7***	0.67*	0.67*	0.57*	-0.19	0.66*
jFace	**0.77***	0.61*	0.65*	0.55*	0.02	0.61*
JDT DB Jdi	**0.75***	0.44*	0.37*	-0.21	0.17	0.02
JDT DB Eval	**0.65***	**0.65***	0.49*	-0.15	0.11	0.06
JDT DB Model	**0.7***	0.65*	0.52*	0.32	0.1	0.45*
Resource	**0.75***	0.63*	0.46*	0.33*	-0.18*	0.65*
Team Core	**0.51** *	0.46*	0.32*	0.28*	0.15	0.45*
CVS Core	**0.76***	0.66*	0.51*	0.31*	0.01	0.48*
DB Core	**0.62***	0.58*	0.49*	0.33*	-0.11	0.53*
JFace Text	**0.75***	0.66*	0.6*	0.56*	-0.16*	0.71*
Update Core	**0.72***	0.64*	0.41*	0.23*	-0.06	0.48*
DB UI	**0.61***	0.56*	0.49*	0.29*	-0.08	0.35*
JDT DB UI	**0.54***	0.48*	0.37*	0.32*	-0.13*	0.31*
Help	**0.47***	**0.47***	0.47*	0.29*	-0.27*	0.3*
JDT Compiler	**0.84***	0.75*	0.52*	0.25*	0.17*	0.51*
JDT Dom	**0.85***	0.73*	0.39*	0.28*	-0.23*	0.46*
JDT Model	**0.73***	0.68*	0.51*	0.23*	-0.05	0.49*
JDT Search	**0.76***	0.61*	0.61*	0.35*	-0.08	0.42*
OSGI	**0.56***	0.52*	0.41*	0.29*	-0.15*	0.49*
Azureus 3	**0.77***	0.67*	0.52*	0.55*	-0.17*	0.67*
Median	**0.73**	0.64	0.49	0.3	-0.08	0.48

project. For more than half of the projects it shows values of 0.7 and above. CBO has the second strongest correlation with a median of 0.64. It still has a substantial correlation on average, however, it is significantly lower than WMC. LCOM and RFC have a median correlation below 0.5. DIT shows a weak correlation with a median of 0.3. NOC shows no correlation with *#SCC* and will be excluded from the prediction experiments.

When comparing the object-oriented metrics with the network centralities, we observe that the median values of WMC and *nOutDregree*—both have the highest median correlation in their respective metric set—differ by 0.07

towards WMC. A *Wilcoxon Signed Rank Test* showed that this difference is significant. Not surprisingly, CBO showed no significant difference with *nOutDegree*, *nInDegree*, and *nPower*; according to their definitions in Table 5.1 and 5.2, these metrics measure the immediate relation to other source files.
To summarize our results: The degree centrality measures *nOutDegree*, *nInDegree*, and *nPower*, and the object oriented metrics WMC and CBO showed substantial to strong correlation with #*SCC* measured for source files. WMC showed the strongest correlation with a median of 0.73. Furthermore, centrality measures based on outgoing connections are significantly stronger correlated than measures based on incoming connections. This is similar to the findings in [ZN08] where outgoing connections of the dependency graph were more related to bugs than incoming connections. In both metric sets we observed measures that show very weak or no correlations at all. Based on these findings we accept **H 1**—*OOM* and *SNA* correlate positively with #*SCC*.

5.3.3 Predicting Change Type Categories

We first performed a series of classification experiments to investigate if network centrality measures and object-oriented metrics can be used for computing models to predict *change-prone* files. We then analyze whether those classification models can be refined towards our change type categories (**H 2**).
Experimental set-up: Prior to classification, we binned the files of each project into *change-prone* or *not change-prone* using the *median* of the total number of *SCC* per file (#*SCC*). These bins represent the *observed classes* when assessing the performance of the classification models later on. The median is a more robust measure, especially for highly skewed values as in our case.

Regarding the calculation of the classification models, we followed the advice by Lessmann *et al.* who found that more sophisticated classifiers might outperform others [LBSP08]. We therefore selected the 8 different classifiers. So far we could not consistently observe significant differences between all those classifiers in our experiments. However, Neural Network (NN) and Bayesian Network (BNet) achieved slightly better results. Hence, we only

Table 5.7: Median classification results over all projects per classifier and per model

	SNA			OOM			SNA&OOM		
	AUC	P	R	AUC	P	R	AUC	P	R
BNet	0.86	0.84	0.78	0.86	0.86	0.74	0.88	0.87	0.84
NN	0.87	0.88	0.82	0.86	0.89	0.81	0.86	0.87	0.83

report on the classification performance of these two learners.

Concerning the evaluation of the classification models, we use the area under the receiver operating characteristic curve statistic (AUC), and also report the median precision (P) and recall (R) values. AUC represents the probability, that, when choosing randomly a *change-prone* and a *not change-prone* file the trained model then assigns a higher score to the change-prone file [LBSP08]. AUC is a robust measure to assess and compare the performance of classifiers since it is independent of prior probabilities and is also the recommended performance measure in [LBSP08, MGF07]. All models were trained using 10 fold cross-validation and AUC, precision, and recall were computed by evaluating each model on the same dataset it was computed from. We consider AUC values above 0.7 to have adequate classification power [LBSP08]. We used Rapid Miner for all prediction experiments.[2]

Discriminating change-prone and not change-prone files: The first set of experiments is concerned with calculating models that are able to classify source files into *change-prone* and *not change-prone*. Table 5.7 lists the median AUC, precision, and recall values across all projects when discriminating files into *change-prone* and *not change-prone*. *SNA* and *OOM* refer to the models that were trained when using centrality measures and object-oriented metrics separately as independent variables. *SNA&OOM* refers to the combined model using both metric sets in combination.

From the median performance values in Table 5.7 we see that the AUC values of the trained models are all above the limit of 0.7, hence show adequate

[2]http://rapid-i.com/

classification power. BNet based on *SNA&OOM* shows the best performance with a median AUC of 0.88, a median precision of 0.87, and a median recall of 0.84. Similarly good results are obtained by NN.

For SNA and OOM separately and for the joint model all AUC values are close. This means that centrality measures as well as object-oriented metrics can predict changes in files equally well. The combination of both metric sets improves the prediction performance slightly but not significantly. A comparison of the values at project level showed that in cases where centrality measures have a comparably lower correlation with changes, prediction models can gain more from using object-orient metrics. For example, JDT DB Eval and DB Core show the largest correlation difference between *SNA* and *OOM* (see Table 5.5).

Discriminating change-prone and not change-prone files according to change type categories: Based on the promising results from the first set of experiments, we next refine our models in order to classify source files into *change-prone*$_C$ and *not change-prone*$_C$ given a change type category C defined in Table 5.4. Analogously to the previous experiment, we binned the files of each project into the observed classes *change-prone*$_C$ and *not change-prone*$_C$ using the median of the total number of changes for a given category C in a file: *cDecl*, *func*, *oState*, *mDecl*, *stmt*, *cond*, or *else*.

We trained Bayesian Networks (BNet) and Neural Networks (NN) in this experiment using *SNA* and *OOM* in combination as input variables. We could not observe a consistently significant performance difference across all categories between both classifiers. However, NN showed a better performance in the case of *cDecl*. We therefore only report on the median results of NN in Table 5.8 and skip the results of BNet for readability and space reasons.

Except for *cDecl* all other categories have an average AUC value above 0.7. A *One Sample Wilcoxon Signed-Ranks Test* showed that except for *cDecl* all other categories have significantly higher median AUC values than 0.7. *oState*, *stmt*, *cond*, and *else* perform significantly higher than 0.8.

The median AUC indicates that prediction performance might vary based on how changes affect source code entities: Categories that aggregate changes

Table 5.8: Median AUC, precision, and recall of across all projects and per category based on Neural Networks (NN)

Project	cDecl	func	oState	mDecl	stmt	cond	else
Median AUC	0.69	0.81	0.84	0.78	0.89	0.86	0.87
Median Precision	0.71	0.82	0.77	0.82	0.9	0.72	0.71
Median Recall	0.73	0.77	0.86	0.77	0.87	0.89	0.88

within method bodies (MB categories), *i.e.*, *stmt*, *cond*, and *else* are above 0.8 and rank as the top three regarding the median AUC values; class body changes (CB categories), *i.e.*, *func* and *oState* have median AUC values close to 0.8; and declaration changes (D categories), *i.e.*, *cDecl* and *mDecl* show the lowest median AUC values around 0.7. We used a *Related Samples Friedman Test* and post-hoc tests with the *Bonferroni Procedure* to statistically investigate the AUC values of the different categories. The tests confirmed aforementioned observations: Within MB, CB, and D the performance differences of the respective change type categories are not significant. However, there are significant differences across MB, CB, and D.

To summarize our results: The first set of experiments showed that SNA and OOM metrics can be used to train models to accurately identify *change-prone* source files. With a median AUC of 0.88, BNet with *SNA* and *OOM* metrics as predictor variables showed the best performance. Second, we refined our models to identify *change-prone* source files according to a given change type category. For each change type category, except *cDecl*, the NN learner obtained models with good predictive power using the *SNA* and *OOM* metrics as independent variables. An analysis of the AUC values among these categories revealed that the performance of the models differs between the types of changes that affect the declaration or body of a class or method. Based on these findings we accept **H 2**—*SNA*&*OOM* measures can predict categories of source code changes.

5.3.4 Manual Analysis of Changes

The correlation analysis and our prediction models showed that coupling to other classes is strongly related with changes. Method invocation statements are part of the *stmt* category and typically account for most of the coupling. To further investigate this potential relationship between coupling and changes, we carried out an initial analysis in which we manually analyzed a sample set of methods and their changes in our dataset. The goal is to find evidence that outgoing dependencies (*i.e.*, method invocations) are indicators for the change-proneness of a method and class, respectively. In particular, we searched for specific instances of invocation statements that changed multiple times in a series of revisions. Such invocations caused maintenance effort over an extended time period rather than only once.

We selected different methods from each project according to the following two criteria: (1) The method was changed frequently in the past (relative to the other methods in the project). (2) A relatively large portion of the changes in a method affected method invocations. After having selected the candidate methods, we manually inspected all subsequent revision-pairs of each method (in total over 350 revisions).

In the following, we report on three representative method examples that we found during the manual inspection. These examples include methods of the classes LaunchConfiguration, ComparePreferencePage, and EclipseSynchronizer of the projects DB Core, Compare and CVS Core, respectively. All these classes exhibit large values of the metrics *nOutDegree* and CBO compared to the project specific median values. Moreover, our BNet model classified those classes as *change-prone* with a probability of 1.0 (see Table 5.9).

In the remainder, $R[r_t - r_{t+1}]$ denotes the subsequent (file) revision pair in which a method changed, *e.g.*, R[1.6-1.7]. For each change we state the change type category in brackets.

LaunchConfiguration.launch(...):[3] 26% of all changes over 20 revisions were method invocation changes. Between revisions R[1.18-1.19] the method call

[3] org.eclipse.debug.internal.core.LaunchConfiguration.launch (String, IProgressMonitor)

Table 5.9: *nOutDegree*, CBO, their medians at project level, and the probability by which BNet models using *SNA* and *OOM* as predictors correctly classified a file as *change-prone*.

File	nOutDegree	Median	CBO	Median	BNet Prob
LaunchConfiguration	0.3	0.003	19	2	1.0
ComparePreferencePage	0.1	0.007	25	2	1.0
EclipseSynchronizer	0.9	0.04	32	6	1.0

`initializeSourceLocator(...)` was added to the method body (1 x *stmt*). In the following, this method invocation changed in 7 revisions. Between R[1.20-1.21], the invocation statement was moved to an if-statement that performs null-checks of its parameters (1 x *stmt*). From R[1.22-1.23] the invocation statement was moved to the else-part of the parent if-statement (1 x *stmt*, 1 x *else*). From R[1.41-1.42] the condition of the if-statement was changed (1 x *stmt*, 1 x *cond*). From R[1.45-1.46] the invocation was moved to a different location within the method body and its else-part was deleted (1 x *stmt*, 1 x *else*). For a better exception handling it was then moved into a try-catch block between R[1.46-1.47] (1 x *stmt*). From R[1.51-1.52] the method invocation was removed and then re-inserted at the same source location in the subsequent revision R[1.52-1.53] (2 x *stmt*). After R1.53, it was not changed anymore.

ComparePreferencePage.createGeneralPage(...):[4] 62% of all changes over 18 revisions were method invocation changes. In this example, a group of similar invocation statements experienced multiple changes over several revisions. From R[1.11-1.12] a new instance of the method invocation `addCheckBox(...)` was added next to two existing ones (1 x *stmt*). It was the beginning of a series of insertions and deletions of instances of this particular invocation spanning over 13 out of 18 revisions of the method. In particular, from R[1.51-1.52] a "commented" instance of the method invocation was inserted (1 x *stmt*); the comment was removed from R[1.52-1.53] (1 x *stmt*).

EclipseSynchronizer.endOperation(...):[5] 30% of all changes over 14 revisions

[4] org.eclipse.compare.internal.ComparePreferencePage.createGeneralPage(Composite)

[5] org.eclipse.team.internal.ccvs.core.resources.EclipseSynchronizer.endOperation(IProgressMonitor)

were method invocation changes. This is an example where even after refactoring changes occurred. From R[1.12-1.13] existing source code was refactored into the new method `commitCache()` and replaced by a corresponding delegate invocation and an if-statement (*1 x stmt*, 1 x *func*). Between R[1.16-1.17] status handling was added to the invocation statement (1 x *stmt*). It was then moved into a try-finally block in R[1.20-1.21] (1 x *stmt*). The condition expression of the if-statement around the invocation and the finally-part were changed from R[1.21-1.22] (1 x *cond*, 4 x *stmt*). Again, the condition expression was changed from R[1.59-1.60] (1 x *cond*, 1 x *stmt*). The finally-part was changed again one revision later, *i.e.*, R[1.60-1.61] (1 x *stmt*).

To summarize our results: All three files of the above described methods exhibit a large number of method invocations in our dataset, *i.e.*, they exhibit *nOutDegree* and *CBO* values significantly above the median of their respective project and were all correctly classified as *change-prone* by our models. These findings support the meaningfulness of our models for predicting *change-prone* files based on their (outgoing) coupling properties. However, a more in depth analysis over time is necessary to validate if an early awareness of the upcoming changes in a particular file raised by our models could have prevented the above observed changes. For instance, in case of the second example by redesigning the initial UI design and behavior.

In this manual analysis we focused on method invocations for two reasons: (1) *nOutDegree* showed the strongest correlation out of all centrality measures. (2) Using CHANGEDISTILLER we can map changes directly to invocation statements as they are part of the AST. Complexity (WMC) on the other hand is based on the control flow graph. Hence, relating fine-grained changes to the concept of *complexity* is less clear.

5.4 Discussion

In the following we discuss the possible scenarios and practical implications that emerge from being able to predict the type of changes (rather than changes

in general) in the context of the software development process, testing, and release planning.

Software development process. The additional information provided by our models about change type categories can help developers in systematically classifying the predicted change-prone files according to the expected change impact and development effort. For example, statement changes (*stmt*) are locally limited to their proximate context and typically do not induce changes at other locations in the source code. In contrast, changes in the API, *e.g.*, denoted by the method declaration (*mDecl*) and functionality (*func*) change type categories, typically have a higher impact and induce more work. For instance, changes in the API also require developers to update the API and design documentation, and to synchronize with other developers using the API. Hence, the ability to predict the "type" of changes that will occur in source files helps to estimate and anticipate in advance the kind and dimension of effort related to that change type.

Testing. Predicting the type of changes is of practical interest for software testing as different changes require different tests: While for small changes (*stmt*) unit-tests are mostly sufficient, API changes, such as indicated by the categories *mDecl* and *func*, might require integration-tests which are more expensive. Adding an else-part (*else*) or changing conditional expressions (*cond*) require new branch-testing procedures that cover the modified structure of the control flow graph. Hence, knowing which types of changes will most likely occur in a source file can help to optimally plan and develop tests, and (in case of limited resources) prioritize among different types of testing.

Release planning. Early awareness regarding the type of changes can help to plan development tasks when facing upcoming release dates. Typically, statement changes are more frequently integrated through small patches, whereas API changes are only released in major steps or not until the final agreement of quality assurance or senior developers.

Furthermore, by applying our models to the source code of a legacy system, they can warn developers if a certain critical threshold for *nOutDegree* or *WMC* is reached. For example, in our dataset the median of these two metrics is

roughly such a threshold regarding the change type category *cond*. Hence, exceeding it will significantly raise the probability of a file to be affected by that category. Based on our models we can provide such simple rules of thumb to developers. This explicit empirical quantification between a particular category of changes and the coupling and complexity structure of a file enables systematic re-factorings to prevent specifically such control flow changes in the future rather than (textual) changes in general.

The basic tools, such as CHANGEDISTILLER and EVOLIZER, exist. As future work we plan to integrate them together with our models into Eclipse or a continuous integration environment, *e.g.*, Hudson, allowing automated model building, *e.g.*, during work, nightly builds, or before committing a new version.

5.5 Threats to Validity

Despite the promising results a careful discussion of the validity of our work is needed.

Our correlation analysis and prediction models indicate the potential existence of relations of *OOM* and *SNA* with #*SCC*, but do not provide a solid causal proof and explanation for it. There might be other reasons that impose changes to a software system as indicated, for instance, by commit messages, bug reports, or patches. Additional work is needed that includes these potential change indicators as well. Moreover, we treated this relation as directional in our work, *i.e.*, using *SNA* and *OOM* to predict #*SCC*. However, the examples in Section 5.3.4 indicate that it might be mutual. For instance, when inserting an invocation statement (*stmt*), the coupling structure is altered by changes themselves. This possibly threatens the content validity. We chose OOM and SNA since they proofed their usefulness in prior work for various prediction tasks. We are aware that other features, *e.g.*, LOC, and feature selection methods, *e.g.*, *Information Gain* [MGF07], exist in literature. For that, further experiments are needed to guarantee optimal performance and models with

the most predictive (sub-)set of features. An extensive comparison study with a richer set of features is part of our future work.

A threat to external validity stems from the fact that our work is dominated by data from the Eclipse platform. This imposes a certain bias caused by characteristics typical to the Eclipse maintenance process, *e.g.*, specific commit behavior. Therefore, the generalizability of the findings and conclusions of this paper might be influenced and have to be verified for other projects. As a matter of fact, any result from empirical work is threatened by the bias of its datasets [MGF07]. Software development is influenced by a variety of factors, both internally and externally, that differ across domains and projects. However, Eclipse and Azureus are relatively large and established systems; both projects have been subject of numerous studies, *e.g.*, [SZZ06, ZPZ07, MPS08, NAH10, GPG11, KR11], before. As such we can benefit from and continue upon prior findings. Replication is central to empirical research and can—even if not carried out identically—enhance existing knowledge [JV09]. Therefore, we are convinced that our findings can add to existing work, sustain current hypotheses and raise new results.

A threat to construct validity might be caused by how we measured *SNA* and *OOM* compared to *SCC*. *SCC* reflect the maintenance activities during a given period. In contrast, *SNA* and *OOM* are based on the source code and represent the dependency and complexity structure at a specific point in time. For this study, the dataset was composed of all *SCC* regarding the timeframes in Table 5.3. On the other hand, *SNA* and *OOM* were measured on the latest source code version available at the end of those timeframes. Our method does not take into account this time gap, *i.e.*, that the relation between *SNA* and *OOM* with *SCC* can change over time.

5.6 Related Work

This section discusses related work about social network techniques in software engineering and change prediction.

Social Network Analysis. Work on this subject stems from the emerging perception that todays software projects are complex networks in which people, technologies, and artifacts show manifold interactions. The idea is to shift to the socio-technial aspects of a project.

Bird *et al.* found out that sub-communities can emerge among the members of open-source (OS) projects [BPD+08]. Social network analysis was applied to CVS information to investigate the structure and evolution of OS projects [GBR04]. A study about the process of people joining OS projects was carried out in [BGD+07].

Our approach focuses solely on source code and its dependency structure rather than the relation of people participating in a certain project. Closer to our work are studies that relate social network analysis to the quality of a software system. In [ZN08], social network measures based on the static dependency graph of Windows Server 2003 binaries turned out to be good indicators for defect-prone binaries. Recently, this work was replicated with data from the Eclipse platform [NAH10]. Pinzger *et al.* related centrality measures computed on the developer contribution network of Windows Vista to post-release failures [PNM08]: More central binaries tend to be more failure-prone. In contrast, our goal is to predict categories of changes rather than defects.

Change Prediction. Rombach was among the first to study the relation between the structure of a system and maintenance [Rom87]. In particular, they showed that architectural design measures capturing the interconnectivity between components influence maintainability. The same object-oriented metrics as in our work were used to predict maintenance in terms of lines changed [LH93]. The results show that these metrics can significantly improve prediction model compared to traditional metrics, *e.g.*, number of semicolons. Their dataset was re-applied in [ZL07] with the focus on comparing the performance of several learning models. Object-oriented metrics and *SCC* were not only successfully applied for maintenance but for defect prediction as well, *e.g.*, [BBM96, SK03, GPG11]. In [KS94] an approach is presented to predict maintenance measured as lines changed using a regression model and a neural

network based on size and complexity metrics. A study to predict different maintenance types, *e.g.*, preventive maintenance, is given in [DJ03]. Similar to our observations, the study states that (immediate) coupling metrics are good predictors while inheritance related metrics are not. To introduce an analogy to meteorology the term *Yesterdays Weather* was coined [GDL04]: Classes that changed recently will likely change again in the future. Dynamic instead of static coupling information was used to predict changes in terms of lines changed [ABF04]. In [MW00] and [KWZ08], two approaches were presented that focus on the probability that a source code change will introduce a failure.

A complementary branch of change prediction is the detection of *change-couplings* between code entities. Such dependencies are often logical and implicit and can not be detected by static analysis alone. Shirabad *et al.* used a decision tree to identify files that are change-coupled [SLM03]. They showed that models built on text features, *i.e.*, words extracted from source code comments and problem reports, performed the best. Tsantalis *et al.* calculated the change-proneness of a class by determining the probability by which it is affected when features in a system change [TCS05]. The idea is to quantify the quality of an object-oriented design that ideally should not be sensitive to changes. Ying *et al.* used association rule mining to recommend additional classes that are potentially relevant for modification tasks [YMNCC04]. The ROSE tool suggests change-coupled source code entities to developers at a fine-grained level, *e.g.*, instance variables [ZWDZ04]. Robbes *et al.* used fine-grained source changes to detect several kinds of distinct logical couplings between files [RPL08]. CHANGEDISTILLER was used to detect changes that are irrelevant (non-essential) to change tasks [KR11]. Our work is complementary in the way that we explored the feasibility to predict categories of changes.

5.7 Conclusions & Future Work

We showed that centrality measures from social network analysis (*SNA*) computed on the static source code dependency graph and object-oriented metrics

(*OOM*) positively correlate with fine-grained source code changes (*SCC*). Our models can output a list of the potentially *change-prone* files ranked according to their change-proneness, overall and per change type category. In summary, the results of our work are:

- *SNA* and *OOM* positively correlate with #*SCC*. Moreover, *Degree Centrality* measures and complexity (WMC) show particularly strong correlations (accepted **H 1**).
- Neural Networks based on *SNA* and *OOM* can predict categories of code change types. For instance, the model for predicting changes in method declarations of classes obtained a median precision of 0.82 and a recall of 0.77 (accepted **H 2**).
- A manual analysis of a subset of changes confirmed the empirical findings regarding the relation between coupling and changes.

Re-running our experiments with different timeframes, *e.g.*, release based, could give insights into how the relation between the position in the dependency graph of a file, its complexity and fine-grained changes evolves over time. Currently our dataset is Eclipse dominated. We plan to replicate our study with other systems including other labeling values in addition to the median. Our models were successful by means of quantitative criteria, *e.g.*, AUC. As future work we will conduct user studies to validate their usefulness for software maintenance, *e.g.*, preventing changes.

6

Classifying Fast and Slowly Fixed Bugs in Open Source Projects

Predicting the Fix Time of Bugs
E. Giger, M. Pinzger, H. Gall
Proc. Int'l Workshop on Recomm. Syst. for Softw. Eng. (RSSE), 2010
DOI: http://doi.acm.org/10.1145/1808920.1808933

Abstract

Two important questions concerning the coordination of development effort are which bugs to fix first and how long it takes to fix them. In this paper we address the latter by empirically investigating the relationships between bug report attributes and the time to fix. The objective is to compute prediction models that can be used to estimate whether a bug will be fixed fast or will take more time for resolution. We use decision tree analysis to compute and 10-fold cross validation to test prediction models. In particular, we explore prediction models in a series of empirical studies with bug report data of six systems of the three open source projects Eclipse, Mozilla, and Gnome. Results show that our mod-

els computed with bug report attributes perform significantly better than random classification. For example, fast fixed Eclipse Platform bugs were classified correctly with a precision of 0.654 and a recall of 0.692. We also show that the inclusion of post-submission bug report data of up to one month can further improve prediction models. The best performing models were obtained with bug report attributes measured at days 7 or 14 after bug reports were opened.

6.1 Introduction

Several open source projects use issue tracking systems, such as Bugzilla, to enable an effective development and maintenance of their software systems. Typically, issue tracking systems collect information about system failures, feature requests, and system improvements. Based on this information and actual project planing, developers select the issues to be fixed. Which issues are addressed first is up to the developers as illustrated by the comment made by J. Arthorne of the Eclipse open source project: *"For most committers, there is an effectively unlimited supply of available bug reports and enhancement requests. How they choose what to work on is effectively a cost/benefit analysis. ... If the cost/benefit ratio of a bug does not change, it may remain unaddressed forever."*[1]

In this paper we aim at investigating prediction models which support developers in the cost/benefit analysis. We address the research question whether we can classify incoming bug reports into fast and slowly fixed. Our approach to answer this question is to look back in the bug reporting history of software projects and empirically analyze the relationships between bug report attributes and the time to fix. In particular, we investigate whether certain attributes of a newly reported bug have an effect on how long it takes to fix the bug. We further investigate, whether prediction models can be improved by including post-submission information which is available within 1 to 30 days after a bug was reported. The two hypotheses of our empirical studies are:

H 1: Incoming bug reports can be classified into fast and slowly fixed.
H 2: Post-submission data of bug reports improves prediction models.

We investigate these two hypotheses with bug report data of six software systems taken from the three open source projects Eclipse, Mozilla, and Gnome. For each subject system we compute decision trees with initial bug report

[1] http://wiki.eclipse.org/Bug_Reporting_FAQ

attribute values and attribute values taken at days 1, 3, 7, 14 and 30 after reports were submitted. A similar approach has been successfully used by Hooimeijer and Weimer to classify bug reports into ”cheap” and ”expensive” to triage [HW07].

Decision tree analysis with 10-fold cross validation is used to train and test prediction models. The predictive power of each model is evaluated with precision, recall, and a summary statistic. Concerning hypothesis H1, prediction models computed with the attributes of incoming bug reports show performance improvements of 10 to 20% over random classification. For example, fast fixed Eclipse Platform bugs were correctly classified with a precision of 0.654 and a recall of 0.692. We also show that the inclusion of post-submission data leads to significant performance improvements. This was true in five out of the six subject systems. The best performing models were obtained when computing decision trees with bug report attributes measured at days 7 or 14. The performance of prediction models is, however, moderate when it comes to use them as support for developers in cost/benefit analysis. Only the two models of Mozilla Firefox and Gnome Evolution showed adequate precision and recall to use them as basis for decisions.

The remainder of the paper is structured as follows: Section 6.2 describes the input data set and our analysis approach. We present and discuss the results of our experiments with the six open source software systems in Section 6.3. Section 6.4 presents related work. Section 6.5 concludes this paper and indicates future work.

6.2 Analysis Approach

The main goal of this work is to find a model for estimating whether a bug will be fixed fast or take more time for resolution. Our approach is to train such a prediction model by looking back in history of software systems and relating bug report attributes with the fix-time of past bugs. We next present the bug report attributes and classification technique that we use to train and

test our models.

6.2.1 Bug Report Attributes

In the first step we obtain bug report information from Bugzilla repositories of open source software projects. Issue tracking systems often show severe deficiencies when it comes to accessing the bug data for analysis. These deficiencies make additional data extraction and processing steps necessary which we automized by using EVOLIZER, our software evolution research and analysis platform.[2] EVOLIZER basically emerged from the idea of having a Release History Database (RHDB) [FPG03] that integrates information originating from different software repositories into a single database.

From the RHDB we select bug reports with the resolution `FIXED`. For each bug report the set of attributes listed in Table 6.1 is computed.

Attributes either correspond directly to bug report fields or are derived from them. For instance, `reporter` is a field in Bugzilla. In contrast `hOpened-BeforeNextRelease` is calculated using the `Opened` field and dates of major system releases. Latter dates are entered by the user. Bug attributes that represent textual descriptions, such as the short and long description of bug reports and the comment texts are not considered in this paper. They long for text-analysis, as has been done by Hooimeijer and Weimer [HW07] or Weiss *et al.* [WPZZ07] who addressed similar research questions. In particular, Weiss *et al.* showed that textual analysis did not yield models with reasonable performance, hence, it remains subject to our future work.

Some attributes of a bug report, such as the `reporter` and the opening date (`monthOpened`, `yearOpened`), are entered once during the initial submission and remain constant. Other attributes, such as `milestone` and `status`, are changed or entered later on in the bug handling process. We highlight attributes that remain constant over time in Table 6.1 by an *I* and attributes that can change by a *C*. We use the change history of bug reports to compute the measures marked with *C* at specific points in time. In addition to

[2] http://seal.ifi.uzh.ch/evolizer

Table 6.1: Constant (*I*) and changing (*C*) bug report attributes.

Attribute	Short Description
`monthOpened`, *I*	month in which the bug was opened
`yearOpened`, *I*	year in which the bug was opened
`platform`, *C*	hardware plaform, *e.g.*, PC, Mac
`os`, *C*	operating system, *e.g.*, Windows XP
`reporter`, *I*	email of the bug reporter
`assignee`, *C*	email of the bug assignee
`milestone`, *C*	identifier of the target milestone
`nrPeopleCC`, *C*	#people in CC list
`priority`, *C*	bug priority, *e.g.*, P1, ..., P5
`severity`, *C*	bug severity, *e.g.*, trivial, critical
`hOpenedBefore-`	hours opened before the next release
`NextRelease`, *I*	
`resolution`, *C*	current resolution, *e.g.*, `FIXED`
`status`, *C*	current status, *e.g.*, `NEW`, `RESOLVED`
`hToLastFix`, *I*	bug fix-time (from opened to last fix)
`nrActivities`, *C*	#changes of bug attributes
`nrComments`, *C*	#comments made to a bug report

the initial values we obtain the attribute values of 24 hours (1 day), 72 hours (3 days), 168 hours (1 week), 336 hours (2 weeks), and 720 hours ($\sim$1 month) after a bug report was opened. The change history of bug reports is stored in bug activities that we apply to the last version of a bug report in reverse order. `nrActivities` simply refers to the number of these changes up to a given point in time. `nrComments` is similar but counts the number of comments entered by Bugzilla users up to the given point in time.

The fix-time `hToLastFix` of each bug report is measured by the time between the opening date and the date of the last change of the bug resolution to `FIXED`. We obtain this date by querying the time stamp of the activity which denotes this last resolution change. The set of attribute values is input to our analysis method that we present in the next section.

6.2.2 Analysis Method

Prediction models are computed with the decision tree algorithm of SPSS.[3] For each experiment we binned bug reports into `Fast` and `Slow` using the median of `hToLastFix`:

$$bugClass = \begin{cases} Fast & : & hToLastFix <= median \\ Slow & : & hToLastFix > median \end{cases} \tag{6.1}$$

`bugClass` is the dependent variable with `Fast` selected as target category. The remaining bug measures are used as independent variables in all of our experiments. The median is a more robust measure than the mean when the input data is heavy skewed. Because both bins are of equal size, the prior probability for each experiment is 0.5 which corresponds to random classification.

We configured the SPSS decision tree algorithm using Exhaustive CHAID (CHi-squared Automatic Interaction Detector) growing method [Kas80]. The CHAID algorithm determines the best split at any node by merging categories of a predictor variable if there is no statistically significant difference within the categories with respect to the target variable. This process is repeated until no non-significant categories are found and done for each predictor variable. While this method saves computation time it does not guarantee to find the best split at each node. This issue was addressed by Biggs *et al.* which extended CHAID by an exhaustive search algorithm [BVS91]. The search algorithm finds the best split by merging similar pairs continuously until only a single pair remains. The set of categories with the largest significance is considered the best split for that predictor variable. We used the default settings of 100 for the minimum number of cases for parent nodes and 50 for the minimum number of cases in leaf nodes. The tree depth was set to 3 levels.

[3] http://www.spss.com/

For the validation of each prediction model we used 10-fold-cross validation [Koh95]. The data set is broken into 10 sets of equal size. The model is trained with 9 data sets and tested with the remaining tenth data set. This process is repeated 10 times with each of the 10 data sets used exactly once as the validation data. The results of the 10 folds then are averaged to produce the performance measures. We use precision (P), recall (R), and the area under the receiver operating characteristic curve (AUC) statistic for measuring the performance of prediction models.

Precision and recall were computed based on the following classification table:

		predicted	
		Slow	Fast
observed	Slow	TN	FP
	Fast	FN	TP

Precision (P) denotes the proportion of correctly predicted `Fast` bugs: $P = TP/(TP + FP)$. Recall (R) denotes the proportion of true positives of all `Fast` bugs: $R = TP/(TP + FN)$. Random classification obtains a precision and recall of 0.5 An ideal model has a precision and recall equal 1.0 which means all bugs were classified correctly. High recall is preferred over high precision because our models aim at identifying the bugs which can be fixed within shorter time.

The receiver operating characteristic (ROC) is a technique to compare two operating characteristics namely the fraction of true positives with the fraction of false positives as the criterion changes [GS66]. The predicted probability of bugs to be classified as `Fast` is selected as the criterion for our experiments. Area under the ROC curve (AUC) represents a summary statistic of an ROC curve. It can be interpreted as the probability, that, when randomly selecting a positive and a negative example the model assigns a higher score to the positive example. In our case the positive example is a bug classified `Fast`.

The higher the probability the better is the performance of a model. Random classification obtains an AUC of 0.5. An ideal model has an AUC of 1.0. Similar to precision and recall this means that the model classified all bugs correctly.

Decision trees for training and testing prediction models as well as precision, recall, and AUC to evaluate prediction models have been used in a number of previous studies with bug report data, for example [KAB+96], [KPB06]. In a recent paper Lessmann *et al.* showed that decision trees and ROC present adequate analysis techniques for this kind of experiments [LBSP08]. Decision trees were only slightly outperformed by random forest classification [Bre01]. On-going work is concerned with investigating whether random forest classification can improve our models.

6.3 Experiments

We investigated the relationships between the fix-time of bug reports and their attributes with six (sub-)systems taken from the three open source software projects Eclipse, Mozilla, and Gnome.[4] Eclipse is an open source platform that offers its functionality through various purpose tailored plugins. Eclipse Platform is a collection of plugins that provide the core framework and services of Eclipse, *e.g.*, resource management and platform user interface. The Java Development Tools (JDT) cover plugins that provide support for developing Java applications with Eclipse, *e.g.*, code completion and debugging facilities. Mozilla is a suite of internet applications. Mozilla Firefox is the internet browser project of Mozilla. In the Bugzilla repository Firefox covers components specifically related to the Firefox browser, *e.g.*, tabbed browsing and phishing protection. Mozilla Core provides basic browser components, *e.g.*, HTML/DOM parsers and a layout engine for HTML web-sites. GStreamer is a multimedia framework incorporated into the Gnome desktop project. Evolution offers integrated mail, address book and calendaring functionality to

[4]The selected (sub-)systems may not be directly mapped one-to-one to concrete products. They rather reflect the organization of a particular Bugzilla repository.

users of the Gnome desktop. Table 6.2 lists the number of bugs input to our experiments.

Table 6.2: Number of bugs and dates of first and last filed bug reports of subject systems.

Project	#Bugs	Observation Period
Eclipse JDT	10,813	Oct. 2001 – Oct. 2007
Eclipse Platform	11,492	Oct. 2001 – Aug. 2007
Mozilla Core	27,392	Mar. 1997 – June 2008
Mozilla Firefox	8,899	Apr. 2001 – July 2008
Gnome GStreamer	3,604	April 2002 – Aug. 2008
Gnome Evolution	13,459	Jan. 1999 – July 2008

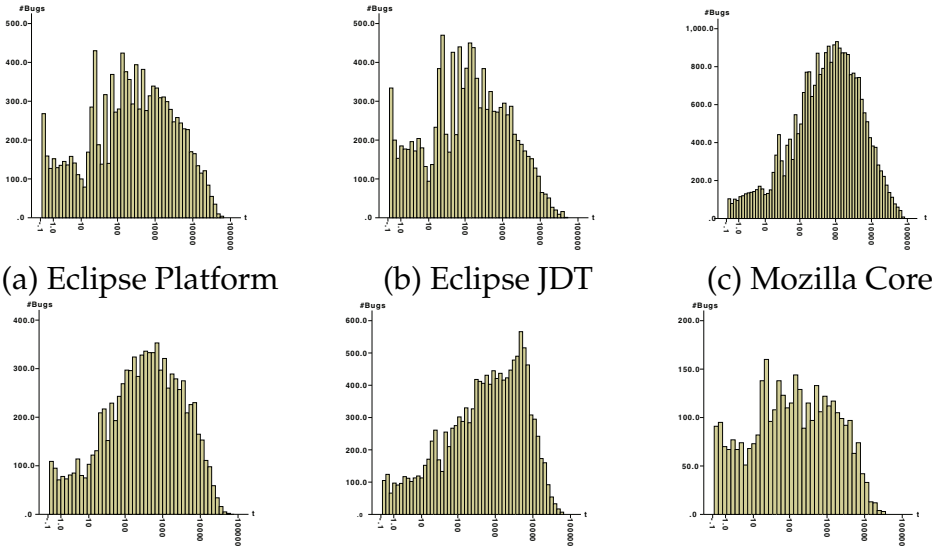
Each case study exhibits a development history of more than six years and provides a significant amount of bug reports. Mozilla Core is the largest case study with more than 27k bug reports back to March 1997. In the following we analyze the distribution of `hToLastFix` to motivate our analysis approach.

6.3.1 Distribution of Fix-Time

The distribution of `hToLastFix` of each system is heavy left-skewed as it is typical for time measures. Most of the bug reports have a short fix-time. Figure 6.1 depicts the corresponding histograms using a logarithmic scale on the x-axis to facilitate visual comparison of distributions.

The histograms of the two Eclipse projects are almost identical. Relatively many bugs were entered and fixed within one hour as indicated by the large bars on the left hand side of the charts. The histogram of the Gnome GStreamer project is of similar shape. In contrast, the histograms of the two Mozilla projects and the Gnome Evolution project show relatively few bugs fixed within one hour.

An analysis of the probability distributions of `hToLastFix` and its log-transformed pendant with Q-Q plots [CCKT83] and One-Sample Kolmogorov-



(a) Eclipse Platform (b) Eclipse JDT (c) Mozilla Core

(d) Mozilla Firefox (e) Gnome Evolution (f) Gnome GStreamer

Figure 6.1: Distribution of `hToLastFix` (log. scale) of bug reports of selected open source systems.

Smirnov tests [CLR67] did not show any significant indicator for normality or normal-like distributions. Based on this results, we categorize bug reports into `Fast` and `Slow` using the median instead of the mean (see Section 6.2.2). Furthermore, we use decision tree analysis which is robust concerning the distribution of values of dependent and independent variables to alleviate this issue. A detailed investigation of the distribution of time-measures is out of scope of this paper.

6.3.2 Classifying Bugs with Initial Bug Data

In this section we present the results of our investigation of hypothesis H1—incoming bug reports can be classified into fast and slowly fixed. Table 6.3 gives an overview of the performance measures obtained by the decision tree analysis. We used `Fast` as target variable for our calculations.

Table 6.3: Performance measures of prediction models computed with initial attribute values.

Project	Median	Prec.	Rec.	AUC
Eclipse JDT	122	0.635	0.485	0.649
Eclipse Platform	258	0.654	0.692	0.743
Mozilla Core	727	0.639	0.641	0.701
Mozilla Firefox	359	0.608	0.732	0.701
Gnome GStreamer	128	0.646	0.694	0.724
Gnome Evolution	701	0.628	0.695	0.694

Eclipse. Looking at Table 6.3 we see that the decision tree model obtained with Eclipse Platform bug reports outperforms the Eclipse JDT model. The most important attribute in the Eclipse Platform model is `monthOpened`. An investigation of the values, however, yielded no clear trend that bug reports are treated differently during the year. The second attribute attached to the tree is `assignee`. The model performance is significantly higher than random classification which lets us accept hypothesis H1 for Eclipse Platform.

With a low recall value of 0.485 the Eclipse JDT model strikes out. A recall value lower than 0.5 indicates that the model misses more than half of `Fast` bug reports. Furthermore, the Eclipse JDT model has the lowest AUC value of all examined projects. The top most attribute of the Eclipse JDT decision tree is `assignee`. The overall structure of the tree affirms the moderate performance of the model. Most of the nodes in the decision tree show low performance to distinguish between fast and slowly fixed bugs. We reject hypothesis H1 for Eclipse JDT.

Mozilla. Decision tree models computed with bug reports of the two Mozilla projects show similar performance. The first attribute considered in the decision tree of the Mozilla Core project is `yearOpened`. Bug reports opened after the year 2003 were more likely to get fixed fast with a probability of 0.632. In contrast, bug reports opened before 2001 tend to be classified `Slow` with a probability of 0.639. Bug reports opened between 2001 and 2003 cannot be distinguished sufficiently by `yearOpened`. Additionally, the decision tree

model contains the component of a bug as well as information about the assignee, the operating system (os), and monthOpened. Improvements over random classification are significant and we accept hypothesis H1 for Mozilla Core.

In contrast to Mozilla Core, the Firefox model contains component as the most significant predictor. There is one node predicting perfectly, however, it only covers 0.9% of bug reports. The second most important attribute is the assignee, and in contrast to the Mozilla Core model, the yearOpened attribute of Firefox bug reports is of only minor relevance. Precision, recall, and AUC values let us accept hypothesis H1 for Mozilla Firefox.

Gnome. The prediction models of both Gnome projects improve random classification. The top most attribute of the Gnome GStreamer decision tree is yearOpened. Similar to Mozilla Core older bug reports (*i.e.*, opened before 2005) were likely to take more time to fix than recently reported bugs. The affected component is the second most significant predictor. An investigation of corresponding tree nodes showed that bug reports which affected components related to the plugin architecture of Gnome GStreamer tend to be fixed faster. In particular recent bug reports followed this trend. As in our previous experiments prediction models were improved by including the attributes reporter and assignee. The values for precision, recall, and AUC let us accept hypothesis H1 for Gnome GStreamer.

The decision tree model of Gnome Evolution bug reports contains assignee as first attribute. The attributes on the second level of the tree are hOpenedBeforeNextRelease, reporter, yearOpened, and severity. An investigation of the decision tree did not show any patterns or tendencies, that enable a straight forward classification of bug reports into Slow and Fast. Concerning precision, recall, and AUC the model performs significantly better than random classification. We accept hypothesis H1 for Gnome Evolution.

In summary, decision tree analysis with the initial bug attributes obtains prediction models that for five out of six systems perform 10 to 20% better than random classification. This is a sufficient indicator that we can compute

prediction models to classify incoming bug reports into `Fast` and `Slow` and we accept hypothesis H1.

6.3.3 Classifying Bugs with Post-Submission Data

This section presents the results of the evaluation of hypothesis H2—post-submission data of bug reports improves prediction models. For each bug report we obtained post-submission data at different points in time, namely 1 day, 3 days, 1 week, 2 weeks, and 1 month after the creation date of the bug report. For each observation period we computed decision tree models which we validated with 10-fold cross validation. The following paragraphs present and discuss the results of experiments and performance measures of prediction models.

Eclipse. Table 6.4 lists the median fix-time of bugs and the results of decision tree analysis with bug reports of the Eclipse JDT project.

Table 6.4: Median fix-time and performance measures of Eclipse JDT prediction models.

Days	Median	#Bugs	Prec.	Rec.	AUC
0	122	10,813	0.635	0.485	0.649
1	296	7,732	0.710	0.577	0.742
3	491	6,277	0.693	0.659	0.767
7	865	4,767	0.750	0.606	0.785
14	1,345	3,653	0.775	0.661	0.823
30	2,094	2,615	0.885	0.554	0.806

The inclusion of post-submission information improved the performance of prediction models as indicated by increasing precision, recall, and AUC. In contrast to the initial decision tree, the models built with post submission data obtained `milestone` as the top most predictor. New bug reports rarely have a milestone specified which, in the case of Eclipse JDT, are 36 out of 10,813 bug reports. Within one week the ratio of pending bugs with milestones increased

to 37% and afterwards remained constant. The inclusion of `milestone` led to improved performance of prediction models for the Eclipse JDT project. In addition to `milestone`, the `assignee`, the `reporter`, `monthOpened`, and `yearOpened` represent significant predictors in computed decision tree models. The best performing model takes into account 14 days of post-submission data. Precision, recall, and AUC values of this model are higher as the corresponding values of the initial model. This lets us accept hypothesis H2 for Eclipse JDT.

Table 6.5 lists the performance measures for the Eclipse Platform bugs. Experiments showed similar results as before with Eclipse JDT. On average, bugs in the Eclipse Platform project tend to take longer to fix than in the Eclipse JDT project. This is indicated by a higher median fix-time for the different observation periods.

Table 6.5: Median fix-time and performance measures of Eclipse Platform prediction models.

Days	Median	#Bugs	Prec.	Rec.	AUC
0	258	11,492	0.654	0.692	0.743
1	560	9,003	0.682	0.586	0.734
3	840	7,803	0.691	0.631	0.749
7	1,309	6,457	0.691	0.587	0.738
14	1,912	5,307	0.743	0.669	0.798
30	2,908	4,135	0.748	0.617	0.788

The inclusion of post-submission data of Eclipse Platform bug reports only sightly improved prediction models. As in the decision tree computed with Eclipse JDT bug reports, the `milestone` attribute was selected as the first attribute in the tree. Also in the Platform data, milestones are added in the post-submission phase of bug reports. After one day, milestones were added to 27% of pending bugs. This ratio remained constant for the later observation points. Most of the undecidable bugs do not have any milestone specified. The `monthOpend`, `reporter`, and `assignee` are the other signifi-

cant predictors contained by decision tree models. The model with 14 days of post-submission data performed best. Improvements over the initial model led to the acceptance of hypothesis H2 for Eclipse Platform.

Mozilla. The results of the decision tree analysis with bug reports of the Mozilla Core project are depicted in Table 6.6. The median bug fix-time indicate longer fix times for Mozilla Core than for Eclipse bugs on average.

Table 6.6: Median fix-time and performance measures of Mozilla Core prediction models.

Days	Median	#Bugs	Prec.	Rec.	AUC
0	727	11,377	0.639	0.641	0.701
1	935	10,424	0.708	0.667	0.773
3	1,179	9,524	0.727	0.630	0.770
7	1,617	8,347	0.712	0.697	0.777
14	2,201	7,142	0.757	0.671	0.803
30	3,257	5,716	0.688	0.708	0.746

Mozilla Core models contained `priority`, `milestone`, `assignee`, and `reporter` as significant predictors. `priority` is the first attribute in decision tree models computed with 3 and 7 days of post-submission data. Bug reports with low priority take longer to fix than bugs with higher priority. For example, in the 3-days model 80.7% of 1,255 bug reports with priority `P1` were fixed fast. `milestone` is the most significant predictor in the other models that consider post-submission data. In Mozilla Core few (1.6%) milestones were entered when the bug was reported. This ratio changed to 30% within one day whereas most of the reports were assigned to the "moz" milestone. The ratio steadily increased up to 47% within 30 days after bug report submission. In extension to Eclipse JDT and Platform, the models computed with Mozilla Core bug reports contained also `severity`, the affected `component`, `nrComments`, and `nrActivities`. Prediction models with post-submission data show improved performance, hence, we accept hypothesis H2 for Mozilla Core.

The median fix-time and performance measures of models computed with

Mozilla Firefox bugs are listed in Table 6.7. The median fix-time indicates faster fixes of Mozilla Firefox bugs than Mozilla Core bugs.

Table 6.7: Median fix-time and performance measures of Mozilla Firefox prediction models.

Days	Median	#Bugs	Prec.	Rec.	AUC
0	359	8899	0.609	0.732	0.701
1	587	7478	0.728	0.584	0.748
3	801	6539	0.697	0.633	0.742
7	1176	5485	0.729	0.610	0.759
14	1778	4553	0.680	0.683	0.757
30	2784	3440	0.751	0.748	0.834

The best model was computed with post-submission data of up to 30 days. This decision tree model has a precision of 0.751, a recall of 0.748, and an AUC of 0.834. While in previous prediction models `milestone` or `priority` were selected as the most significant predictors, `nrActivities` and `yearOpened` were selected in Mozilla Firefox models. In the models computed with 1, 3, and 7 days of post-submission data we observed, that bugs with zero or one activity were fixed slower than bugs with more than 7 activities. The ratio of bug reports with specified milestones follows a similar trend as in previous case studies. Surprisingly, the model with the best performance (30 days) does not contain the `milestone` attribute. In this model, `yearOpened` is the most significant predictor. In particular, bugs that were reported before the year 2003 took longer to fix on average than bugs reported after the year 2006. The `reporter` and `assignee` were the other bug attributes contained by this decision tree. The good performance of the last model (30-days) lets us accept the hypothesis H2 for Mozilla Firefox.

Gnome. Table 6.8 lists the measures of the prediction models computed with the Gnome GStreamer bug reports. Similar to bug reports of the two Eclipse projects many reports in Gnome GStreamer have a short fix-time on average as indicated by lower median fix-time.

Table 6.8: Median fix-time and performance measures of Gnome GStreamer prediction models.

Days	Median	#Bugs	Prec.	Rec.	AUC
0	128	3604	0.646	0.694	0.724
1	406	2553	0.581	0.810	0.666
3	708	2052	0.606	0.704	0.667
7	1084	1650	0.613	0.652	0.669
14	1517	1351	0.658	0.561	0.680
30	2268	1018	0.538	0.811	0.586

In contrast to previous experiments, the performance of models computed for Gnome GStreamer decreases with the inclusion of post-submission information. While the AUC value of the initial model is 0.724 the AUC value of the last model is only 0.586. One big difference is that in Gnome GStreamer the `milestone` attribute is specified for only few bug reports, hence, was not included into prediction models. Although, milestones were initially specified for 9% of bug reports, this ratio increased to only 18% within 30 days which is lower than the ratio in the Eclipse or Mozilla projects. In the models with post-submission data, the `assignee` is the most significant predictor followed by the `reporter` and `nrComments`. Also with post-submission data we could not obtain reasonable prediction models, hence, we reject hypothesis H2 for the Gnome GStreamer.

The next series of experiments was with bug reports of the Gnome Evolution project. The results of the decision tree analysis are depicted by Table 6.9. Bugs of this system tend to take longer to fix on average than in the other subject systems.

Compared to Gnome GStreamer the models computed with the Gnome Evolution bug reports show better performance regarding precision, recall, and AUC values. The performance of the decision tree models increases when including post-submission information. Similar to the decision trees computed with Eclipse and Mozilla bug reports `milestone` is the most significant predictor followed by `assignee`. Milestones were added for 21% of

Table 6.9: Median fix-time and performance measures of Gnome Evolution prediction models.

Days	Median	#Bugs	Prec.	Rec.	AUC
0	701	13459	0.628	0.695	0.694
1	1136	11548	0.649	0.659	0.727
3	1476	10496	0.693	0.611	0.746
7	1962	9335	0.636	0.798	0.752
14	2566	8228	0.665	0.760	0.766
30	3625	6695	0.690	0.682	0.771

the bugs within one day. This ratio increased to 31% within 30 days. Slow bug reports are indicated by milestones, such as "Later", "Future", or "reschedule" while fast bug reports got mainly concrete release numbers. Bug reports with no milestone are basically undecidable. Other significant predictor variables which appeared in the various Gnome Evolution models are the `reporter`, `yearOpened`, and `monthOpened`. Furthermore, `severity` and `hOpenedBeforeNextRelease` are significant. The good performance of the prediction models with 7 and 14 days of post-submission data lets us accept hypothesis H2 for Gnome Evolution.

In summary, the inclusion of post-submission data led to improved predictive power of models in all systems but Gnome GStreamer. We therefore accept hypothesis H2.

6.3.4 Summary of Results

The results of our experiments can be summarized as follows:

Incoming bug reports can be classified into fast and slowly fixed. Using decision trees with Exhaustive CHAID on the initial values of bug report attributes we computed prediction models that improved random classification significantly. For example the Eclipse Platform model showed a precision of 0.654, a recall of 0.692, and an ROC AUC of 0.743. Based on the recall,

prediction models can classify ∼70% of `Fast` bug reports correctly which denotes an improvement of ∼20% over random classification. The other models, except the one of Eclipse JDT, showed similar performance. Resulting decision trees differed from system to system. Attributes that were included in all decision trees are `assignee`, `yearOpened`, and `monthOpened`. We accepted hypothesis H1 | newly reported bug reports can be classified into fast and slowly fixed.

Post-submission data of bug reports can improve prediction models. The inclusion of post-submission bug report data led to improved performance of prediction models in all projects but Gnome GStreamer. In Eclipse JDT 3 days, in Eclipse Platform 14 days, in Mozilla Core 1 day, in Mozilla Firefox 30 days, and in Gnome Evolution 7 days of post-submission data yielded models which significantly improved the classification of bug reports. For example, in Eclipse JDT the 3-days decision tree model improved the recall of `Fast` bug reports from 0.485 to 0.659. The best-performing prediction models were obtained with 14-days or 30-days of post-submission data. Results showed that bug reports with a concrete `milestone` enabled a more accurate classification. The `assignee`, `reporter`, `monthOpened` and `yearOpened` were also significant predictors that emerged in decision tree models. In contrast to the results of Hooimeijer and Weimer [HW07] the post-submission features `nrCommits` and `nrActivities` showed only minor importance. Except for Gnome GStreamer we accepted H2 | post-submission data of bug reports can improve prediction models.

Application of results. Decision tree models with initial and post-submission bug report data showed adequate performance when compared to random classification. However, the applicability of these models to develop recommender systems, that aid developers in classifying incoming bug reports into `Fast` and `Slow`, is questionable. Most models have a precision and recall between 0.65 and 0.75 which means that 65 to 75% of bug reports are classified correctly. For most of the subject systems such performance can be reached

with prediction models computed with `milestone`, `assignee`, `reporter`, `monthOpened`, and `yearOpened`. Prediction models can provide valuable input to developers and aid them in deciding which bugs to address first, though, can not be solemn based on the output of models.

6.3.5 Threats to Validity

From an *external validity* point of view there are two threats. First, we only chose open source projects. The results obtained in our work may differ for closed source (industrial) systems. We plan to extend our work with closed source projects and compare the results. Nevertheless, we abate this threat by the fact that the projects in our studies are from different domains and are implemented in different programming languages.

Second, according to Figure 6.1 in Section 6.3.1 we can see especially in the Eclipse projects and in GStreamer that many bugs were fixed within a short time. A possible explanation is that developers reported a bug—for the sake of completeness and information—although they had already a fix at hand at that time. This phenomena distorts our data because we can not calculate the actual fix-time of such bugs. We alleviate this threat with large number of bugs in our studies and by using the median of the fix-time to categorize the dataset into two bins of equal size.

Threats to *internal validity* arise primarily from our measure `hToLastFix` presented in Section 6.2. First, in our studies `hToLastFix` is measured at the given point in time when we extracted our data from the repository. It may change, *e.g.*, a bug can be reopened. Second, we take the date of the last change of the bug resolution to `FIXED` to calculate `hToLastFix`. This could be misleading as there might be a timely gap between the date when a bug receives the status `RESOLVED`, `VERIFIED`, or `CLOSED`, *i.e.*, there is a solution available and the date when the bug finally gets tagged as `FIXED`.

Another threat is, that we only included bugs with `FIXED` as resolution. Hence we cannot assume a priori that the results of our study generalize also to bugs having other resolution values, *e.g.*, `DUPLICATE`, `WORKSFORME`, or

WONTFIX.

6.4 Related Work

The field of *Mining Software Repositories* has received much attention in recent years. With the growing number of open source projects that organize an important part of their activities over software repositories, *e.g.*, issue tracking systems, versioning systems, or mailing lists more and more data becomes available to researchers for analysis purposes. The idea is to apply data analysis and mining techniques to data extracted from such repositories and then drawing conclusions and gaining insights about the evolution of a system and its process.

Hooimeijer and Weimer [HW07] used linear regression analysis on bug report data to predict whether a bug report is triaged within a given amount of time. Similar to our approach they take into account post-submission data and investigate how much of this data is needed to yield adequate predictive power. While they focus on reducing the bug triage time which they denote as time needed to inspect, understand, and making the initial decision regarding how to address the report, we concentrate on the fix-time of bugs. Furthermore, they aim at finding an optimal cut-off value to classify bug reports into ”cheap” and ”expensive” while we use fixed cut-off values for `Fast` and `Slow`. Additionally, we use decision tree analysis instead of linear regression analysis.

Panjer used several different data mining models to predict eclipse bug lifetimes [Pan07]. We extend his work by looking at more systems. Furthermore, while he counted the cc list, dependent bugs, bug dependencies, and comments we take into account that other attributes, *e.g.*, `assignee` might change as well. We rather create different profiles representing the state of a bug at a certain point of its lifetime.

An approach to assist in bug triage is presented by Anvik *et al.* in [AHM06]. They give suggestions to which developer a new bug report should be assigned. To find suitable developers among all possible candidates they apply

machine learning techniques to open bug repositories. It would be interesting to see whether vector support machines instead of decision trees can improve prediction in our sense. The same authors provide a study on the character of open bug repositories and describe problems rising when using such repositories, *e.g.*, duplicates or filtering out irrelevant reports [AHM05].

Similarly, Ko *et al.* examined a large number of bug reports and analyzed characteristics of them in [KMC06]. Having an idea of the nature of bug reports and how they are written have implications on tool design and software engineering. This may lead to more consistent bug reporting data which can improve our prediction models.

Wang *et al.* recognize the mentioned problem of duplicates and its possible drawbacks on bug triage. They combine natural language information and execution information of any new arriving bug to find the most similar bugs among already existing ones. Those are then suggested as candidates for duplicates to the developers [WZX+08]. Involving execution information they extend earlier work that focused mostly on natural language processing for detection of duplicate bug reports [RAN07].

Kim and Whitehead argue that the time needed to fix a bug is a significant factor when measuring the quality of a software system [KW06]. Our approach is complementary, in that it provides a prediction model for estimating whether a bug will be fixed fast or take more time for resolution.

Given a new bug report Weiss *et al.* present a method to predict the effort, *i.e.*, the person-hours spent on fixing that bug [WPZZ07]. They apply text mining technique to search reports that match a new filed bug. They use effort measures from past bug reports as a predictor. We also use existing data from recoded bug reports to compute a prediction model but we remain limited to non-textual features of a bug report.

Bettenburg *et al.* investigated which elements developers rely on when fixing a bug [BJS+08]. Similar to our approach they claim that the information given in bug reports has an impact on the fix time. By having a model to describe the quality of a bug report they are able to assist reporters and give feedback on the quality of a new bug report. In our work we do not consider

any quality aspects of the information in a report. Our analysis rather treats such information as given facts and investigates how well it can be used as a predictor. However, our work might benefit from their approach, in such way, that we investigate whether and how the prediction performance and the quality of a bug report is related.

Similar to our work Khoshgoftaar *et al.* use decision trees to predict fault prone modules in a military software system [KAB+96]. Selby and Porter in [SP88] used decision trees to classify software modules that had high development effort. Both approaches achieved good results in applying discussion trees in the software environment. Lessmann *et al.* compare different classification models for software defect prediction using AUC as benchmark [LBSP08]. We use similar analysis techniques and performance evaluation criteria but instead of failure-proneness aim at providing models to predict the fix time of bugs.

Recently Bird *et al.* have found evidence the there is a systematic bias in bug datasets [BBA+09]. This might effect prediction models relying on such biased datasets.

6.5 Conclusions & Future Work

How long does it take to fix an issue is an important questions for the coordination of development efforts in software projects. In this paper we showed that prediction models computed with decision tree analysis on bug report data provide valuable input to developers to answer this question. Our conclusions are based on the results obtained from a series of experiments with bug report data of six systems from three large and active open source projects. We computed prediction models with initial bug report data as well as post-submission information. Summarized, the results of our experiments are:

- Between 60% and 70% of incoming bug reports can be correctly classified into fast and slowly fixed. `assignee`, `reporter`, and `monthOpened`

are the attributes that have the strongest influence on the fix-time of bugs (see Section 6.3.3).

- Post-submission data of bug reports improves the performance of prediction models by 5% to 10%. The best-performing prediction models were obtained with 14-days or 30-days of post-submission data. The addition of concrete `milestone` information was the main factor for the performance improvements. (see Section 6.3.3).

On-going and future work is basically concerned with improving the performance of prediction models. For this we plan to extend the input data set and investigate other algorithms to compute prediction models. For example, detailed change information of bug report attributes and data about the affected components will be tested. Furthermore, we plan to evaluate whether Random Forests and Naive Bayes algorithms can improve prediction models.

7

Conclusions

SOFTWARE prediction models are an effective way to support developers in software maintenance tasks, *e.g.*, bug triaging and bug fixing. By combining statistical methods, data mining approaches, machine learning techniques, and project data they provide an analytical and structured basis to make decisions. For that, software prediction models make use of data that is archived in software repositories, *e.g.*, bug report information and the development history of a software system. The development history and its corresponding source code changes are stored and tracked in Version Control Systems (VCS), *e.g.*, CVS, SVN, or GIT. However, current VCS treat and manage source code as pure text and track changes on file-level, i.e., file revisions. Therefore, such changes suffer from two problems: *Too coarse-grained change information* and *Lack of change semantics*.

For instance, the missing ability to differentiate between pure textual changes, *e.g.*, license header updates and formatting, and actual code changes can potentially interfere the results and accuracy of bug prediction models that operate on traditional code churn metrics only, *e.g.*, lines added/deleted/changed.

To address these shortcomings, research investigated the applicability of alternate, more fine-grained changes to analyze the development history of

software, *e.g.*, [ZWDZ04, KZWZ07, RPL08, KR11, RD11]. In particular, Fluri *et al.* presented an approach to extract fine-grained source code changes based on the structure of the abstract syntax tree (AST) of source code [FWPG07].

We contribute to this research direction by leveraging the benefits of those AST-based code changes for software prediction model building. In particular, we state that using fine-grained source code changes allows us to train more accurate prediction models in terms of (1) prediction performance and (2) prediction granularity, and (3) that take into account the different type semantics of changes, *e.g.*, method declaration changes versus statement changes.

7.1 Summary of Results

The foundation and contributions of this thesis consist of five empirical studies each containing a series of prediction experiments. In these studies we use statistical methods, data mining approaches, and machine learning techniques to analyze software prediction models that are based on (fine-grained) source code changes and bug data extracted from the development history of several open-source projects.

In the following, we summarize the goals and results of each empirical study.

Study 1 (Chapter 2, p. 29 *et seq.*). The focus of this study is comparing fine-grained source code changes and traditional code churn for bug prediction. The findings are:

- Fine-grained source code changes exhibit a significantly stronger correlation with the number of bugs than code churn based on lines modified.

- Classification models using fine-grained source code changes rank *bug-prone* files higher than *not bug-prone* ones with an average probability of 90%. This is a significant improvement compared to models computed with code churn.

- Non-linear asymptotic regression using fine-grained source code changes obtained models to successfully predict the number of bugs with a median R^2 of 0.79 which is a significant improvement over models computed with code churn.

Study 2 (Chapter 3, p. 77 *et seq.*). To goal of this study is to build bug prediction models at the level of individual methods rather than at file-level. Being able to narrow down the location of bugs to method-level can save manual inspection steps and significantly improve testing effort allocation. The findings are:

- Change metrics (extracted from the version control system of a project) can be used to train prediction models with good performance. For example, a Random Forest model achieved an AUC of 0.95, precision of 0.84, and a recall of 0.88.
- Using change metrics as predictor variables produced prediction models with significantly better results compared to source code metrics. However, combining both metric sets did not improve the classification performance of our models.

Study 3 (Chapter 4, p. 105 *et seq.*). We use the Gini coefficient—a popular economic metric—to measure code ownership and relate it to bugs. The findings are:

- The number of bugs in a file correlates negatively with the Gini coefficient: The more changes of a file are done by a few dedicated developers (high Gini coefficient), the less likely it will have bugs.
- The Gini coefficient can be used to identify *bug-prone* files with adequate performance. The best results (AUC of 0.81) are obtained with a Random Forest prediction model.

Study 4 (Chapter 5, p. 121 *et seq.*). We explore prediction models for whether a source file will be affected by a certain type of source code change. The findings are:

- Neural Networks using static dependency information and object oriented CK-metrics as input variables can predict categories of source code change types. For instance, the model for predicting changes in method declarations of Java classes obtained a median precision of 0.82 and a recall of 0.77.
- Coupling to other files and complexity show particularly strong correlations with the number of fine-grained source code changes.

Study 5 (Chapter 6, p. 147 *et seq.*). The objective of this study is to compute prediction models that can be used to estimate whether a bug will be fixed fast or will take more time for resolution. The findings are:

- Between 60% and 70% of incoming bug reports can be correctly classified into fast and slowly fixed. `assignee`, `reporter`, and `monthOpened` are the attributes of a bug report that have the strongest influence on the fix-time of bugs.
- Post-submission data of bug reports improves the performance of prediction models by 5% to 10%. The best-performing prediction models were obtained with 14-days or 30-days of post-submission data. The addition of concrete `milestone` information was the main factor for the performance improvements.

7.2 Implications of Results

In the last decade, software prediction models experienced increased attention in industry and research. In particular, ongoing research is dedicated to improve the performance of bug prediction models and is nowadays driven by a large community. The huge and diverse amount of data being generated by software development, and moreover, the ever-increasing availability of this data for researchers have led to more sophisticated prediction models: They constantly advance our understanding of how the characteristics of the source

code and of its development process are (quantitatively) related to bugs, and, as tools, they provide more effective means to make decisions.

By incorporating fine-grained source code changes and their types into prediction models we can contribute to that existing body of knowledge, strengthen existing hypotheses, and present new results that advance the current state-of-the-art.

The results and findings of our empirical studies give evidence that fine-grained source code changes can improve software prediction models in terms of prediction performance and prediction granularity (compared to traditional change metrics). One the one hand, the finer prediction granularity, *i.e.*, method-level instead of file-level, helps to narrow down the search space for identifying the bug-prone localities of the source code, and hence, spares developers a substantial number of manual inspection steps, *i.e.*, time. Considering that larger files are known to be among the most bug-prone ones, the benefit of having method-level prediction models is even higher. On the other hand, the increased prediction performance can help managers allocating their (limited) resources more efficiently to the bug-prone parts of a system.

One particular strength of our prediction models is that they include the type of source code changes. This closes a major semantic gap of the standard change measures, *e.g.*, (textual) code churn or revisions, as they are rather generic. For instance, the ability to distinguish between indentation and formatting changes, license header updates, small statement changes, and declaration changes can help researchers and developers to improve their interpretations of change-based bug prediction models, and hence, allows for a better understanding of the empirical relation between bugs, changes, and their types — a point that was already raised for change-based analyses in general [KR11]. Furthermore, the awareness which particular change types are more critical with respect to bugs can help to optimally allocate testing resources for quality assurance in a project.

The benefits of combining fine-grained source code changes and prediction models come with the additional effort that is needed to extract these changes from the project history. This is, however, not an issue when tools, such as

CHANGEDISTILLER, are available that perform this extraction fully automatically. There are research efforts to integrate fine-grained source code changes more seamlessly into the development process. For instance, the *Molhado Hypertext Versioning System* provides a tree based, structured versioning approach [NMB04]. *SpyWare*, on the other hand, puts code changes in the center by monitoring development activities in the IDE itself rather than analyzing the change history recorded in the version control systems [RL08].

Nevertheless, it would be of great interest to embed the concept of fine-grained code changes inherently into versioning control systems to release their full potential for future research in the field of Mining Software Repositories.

7.3 Future Work

During our work we found opportunities for further research:

- Most of our studies use data of the change history of open-source projects. Therefore, conclusions made in this work can be biased by characteristics of the development process that are specific and unique to this kind of project organization. This might threaten the generalizability of our results beyond open-source scenarios, *e.g.*, commercially developed software. To address this issue replications of our study with other projects are required.
- We evaluated our software prediction models by means of classification performance measures, *i.e.*, precision, recall, and AUC. However, to provide more evidence regarding the practical benefits user studies are required that validate the effectiveness of our models in the context of real world scenarios.
- Complementary to the previous point is tool support. Currently the basic tools, such as CHANGEDISTILLER to extract the fine-grained source code changes or EVOLIZER to automatically collect bug data [GFP09], exist. Furthermore, data mining frameworks, such as RapidMiner [MWK+06], enable a fully automated data mining and prediction analysis. However,

to make our prediction models useful and applicable in practice a seamless integration of all steps, *i.e.*, data collection, data aggregation, data analysis, result generation, (to some extent) result interpretation, and result visualization within a single tool is necessary. Therefore, we plan to integrate our work into the Eclipse IDE[1] or a continuous integration environment, *e.g.*, Hudson[2].

- The scope of this thesis are empirical and quantitative studies. However, correlation analysis and prediction models only indicate the potential existence of relations between fine-grained source code changes and bugs, but do not provide a solid causal proof and explanation for it. Further research that combines fine-grained source code changes, *i.e.*, quantitative data, with more qualitative data, *e.g.*, commit messages or bug reports, is needed. Such work might reveal latent factors that cause bugs in a software system.

[1] http://www.eclipse.org/
[2] http://hudson-ci.org/

Bibliography

[AAP+08] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement? a text-based approach to classify change requests. In *Proc. Conf. of the center for advanced studies on collaborative research: meeting of minds*, pages 304–318, 2008.

[AB06] Erik Arisholm and Lionel Briand. Predicting fault-prone components in a java legacy system. In *Proc. Int'l Symposium on Empirical Softw. Eng.*, pages 8–17, 2006.

[ABF04] Erik Arisholm, Lionel Briand, and Audun Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Trans. Softw. Eng.*, 30(8):491–506, 2004.

[AHM05] John Anvik, Lyndon Hiew, and Gail Murphy. Coping with an open bug repository. In *Proc. OOPSLA Workshop on Eclipse Technology eXchange*, pages 35–39, 2005.

[AHM06] John Anvik, Lyndon Hiew, and Gail Murphy. Who should fix this bug? In *Proc. Int'l Conf. on Softw. Eng.*, pages 361–370, 2006.

[BBA+09] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proc. Joint European*

Softw. Eng. Conf. and ACM SIGSOFT Symposium on the Foundations of Softw. Eng., pages 121–130, 2009.

[BBM96] Victor Basili, Lionel Briand, and Walcélio Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22:751–761, October 1996.

[BEF99] Steve Borgatti, Martin Everett, and Lin Freeman. *UCINET 5.0 Version 1.00*. Natick: Analytic Technologies., 1999.

[BEP07] Abraham Bernstein, Jayalath Ekanayake, and Martin Pinzger. Improving defect prediction using temporal features and non linear models. In *Proc. Int'l Workshop on Principles of Softw. Evolution*, pages 11–18, 2007.

[BFN+06] Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the shape of java software. In *Proc. ACM SIGPLAN Conf. on Object-oriented programming systems, languages, and applications*, pages 397–412, 2006.

[BGD+07] Christian Bird, Alex Gourley, Premkumar Devanbu, Anand Swaminathan, and Greta Hsu. Open borders? immigration in open source projects. In *Proc. Int'l Workshop on Mining Softw. Repositories*, page 6. IEEE Press, 2007.

[BJS+08] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proc. ACM SIGSOFT Int'l Symposium on Foundations of Softw. Eng.*, pages 308–318, 2008.

[BMW02] Lionel Briand, Walcelio Melo, and Juergen Wuest. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Trans. Softw. Eng.*, 28(7):706–720, July 2002.

[BND+09] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. Does distributed development affect software quality? an empirical case study of windows vista. In *Proc. Int'l Conf. on Softw. Eng.*, pages 518–528, 2009.

[BNM+11] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't Touch My Code! Examining the Effects of Ownership on Software Quality. In *Proc. Joint European Softw. Eng. Conf. and ACM SIGSOFT Symposium on the Foundations of Softw. Eng.*, page to appear, 2011.

[Boe76] Barry Boehm. Software engineering. *IEEE Trans. on Computers*, C-25(12):1226–1241, Dec. 1976.

[Boe81] Barry Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[Bon87] Phillip Bonacich. Power and centrality: A family of measures. *The American Journal of Sociology*, 92(5):1170–1182, 1987.

[BPD+08] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent social structure in open source projects. In *Proc. ACM SIGSOFT Int'l Symposium on Foundations of Softw. Eng.*, pages 24–35. ACM, 2008.

[Bre01] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[Bro95] Frederick Brooks. *The Mythical Man-Month*. Addison-Wesley, anniversary edition, 1995.

[BS98] Aaron Binkley and Stephen Schach. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In *Proc. Int'l Conf. on Softw. Eng.*, pages 452–455, 1998.

[BSL99] Victor Basili, Forrest Shull, and Filippo Lanubile. Building knowledge through families of experiments. *IEEE Trans. Softw. Eng.*, 25:456–473, July 1999.

[BVS91] David Biggs, Barry De Ville, and Ed Suen. A method of choosing multiway partitions for classification and decision trees. *Journal of Applied Statistics*, 18(1):49–62, 1991.

[BWIL99] Lionel Briand, Jürgen Wüst, Stefan Ikonomovski, and Hakim Lounis. Investigating quality factors in object-oriented designs: an industrial case study. In *Proc. Int'l Conf. on Softw. Eng.*, pages 345–354, 1999.

[Car93] Roland Carver. The case against statistical significance testing, revisited. *The Journal of Experimental Education*, 61(4):287–292, 1993.

[CCKT83] John Chambers, William Cleveland, Beat Kleiner, and Paul Tukey. *Graphical Methods for Data Analysis*. Wadsworth, 1983.

[CH96] Dennis Christenson and Steel Huang. Estimating the fault content of software using the fix-on-fix model. *Bell Labs Technical Journal*, 1(1):130–137, 1996.

[CHV09] Harris Cooper, Larry Hedges, and Jeffery Valentine, editors. *The Handbook of Research Synthesis and Meta-Analysis*. Russell Sage Foundation Publications, 2nd edition, 2009.

[CK94] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[CL01] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001.

[CLR67] I. M. Chakravarti, R. G. Laha, and J. Roy. *Handbook of Methods of Applied Statistics*, volume 1. John Wiley and Sons, 1967.

[Coh88] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 2nd edition, 1988.

[DJ03] Melis Dagpinar and Jens Jahnke. Predicting maintainability with object-oriented metrics - an empirical comparison. In *Proc. Working Conf. on Reverse Eng.*, pages 155–164, 2003.

[DKS06] Tore Dyba, Vigdis Kampenes, and Dag Sjoberg. A systematic review of statistical power in software engineering experiments. *Information and Software Technology*, 48(8):745–755, 2006.

[DLR10a] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Proc. Int'l Working Conf. on Mining Softw. Repositories*, pages 31 –41, 2010.

[DLR10b] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Proc. Int'l Workshop on Mining Softw. Repositories*, pages 31–41, 2010.

[DLR11] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empir. Softw. Eng.*, pages 1–47, 2011.

[Dor79] Robert Dorfman. A formula for the gini coefficient. *The Review of Economics and Statistics*, 61(1):146–149, February 1979.

[DP02] Giovanni Denaro and Mauro Pezzè. An empirical evaluation of fault-proneness models. In *Proc. Int'l Conf. on Softw. Eng.*, pages 241–251, 2002.

[Duc05] Nicolas Ducheneaut. Socialization in an open source software community: A socio-technical analysis. *Comput. Supported Coop. Work*, 14:323–368, 2005.

[DWC04] Shirley Dowdy, Stanley Weardon, and Daniel Chilko. *Statistics for Research*. Probability and Statistics. John Wiley and Sons, Hoboken, New Jersey, third edition, 2004.

[EBGR01] Khaled El Emam, Saïda Benlarbi, Nishith Goel, and Shesh Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. on Softw. Eng.*, 27(7):630–650, July 2001.

[EGK+01] Stephen Eick, Todd Graves, Alan Karr, James Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Trans. Softw. Eng.*, 27(1):1–12, January 2001.

[Erl00] Len Erlikh. Leveraging legacy system dollars for e-business. *(IEEE) IT Professional*, 2(3):17 –23, may/jun 2000.

[ETGB] Jayalath Ekanayake, Jonas Tappolet, Harald Gall, and Abraham Bernstein. Time variance and defect prediction in software projects. *Empir. Softw. Eng.*, pages 1–42.

[FG06] Beat Fluri and Harald C. Gall. Classifying change types for qualifying change couplings. In *Proc. Int'l Conf. on Program Comprehension*, pages 35–45, 2006.

[FGG08] Beat Fluri, Emanuel Giger, and Harald C. Gall. Discovering patterns of change types. In *Proc. Int'l Conf. on Automated Softw. Eng.*, page 4, 2008.

[FGP05] Beat Fluri, Harald Gall, and Martin Pinzger. Fine-grained analysis of change couplings. In *Proc. Int'l Workshop on Source Code Analysis and Manipulation*, pages 66–74, 2005.

[FI93] Usama Fayyad and Keki Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proc.*

of the Int'l Joint Conf. on Artificial Intelligence, pages 1022–1027, 1993.

[FN99] Norman Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25:675–689, September 1999.

[FPG03] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proc. Int'l Conf. on Softw. Maintenance*, pages 23–32, 2003.

[Fre79] Linton C. Freeman. Centrality in social networks conceptual clarification. *Social Networks*, 1(3):215 – 239, 1979.

[FWPG07] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. on Softw. Eng.*, 33(11):725–743, November 2007.

[FZG08] Beat Fluri, Jonas Zuberbuehler, and Harald C. Gall. Recommending method invocation context changes. In *Proc. Int'l Workshop on Recomm. Syst. for Softw. Eng.*, pages 1–5, 2008.

[GBR04] Luis González, González Barahona, and Gregorio Robles. Applying social network analysis to the information in cvs repositories. In *Proc. Int'l Workshop on Mining Softw. Repositories*, pages 101–105, 2004.

[GDL04] Tudor Girba, Stephane Ducasse, and Michele Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proc. Int'l Conf. on Softw. Maintenance*, pages 40–49, 2004.

[GFP09] Harald C. Gall, Beat Fluri, and Martin Pinzger. Change analysis with evolizer and changedistiller. *IEEE Software*, 26(1):26–33, January/February 2009.

[GFS05] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Softw. Eng.*, 31:897–910, 2005.

[GG11] Giacomo Ghezzi and Harald Gall. Sofas: A lightweight architecture for software analysis as a service. In *Proc. Working Conf. on Softw. Architecture*, pages 93–102, 2011.

[Gin12] Corrado Gini. Variabilità e mutabilità. *Memorie di metodologica statistica*, 1912.

[GKMS00] Todd Graves, Alan Karr, James Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26:653–661, July 2000.

[GPG11] Emanuel Giger, Martin Pinzger, and Harald C. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proc. Int'l Workshop on Mining Softw. Repos.*, pages 83–92, 2011.

[GS66] David M. Green and John A. Swets. *Signal Detection Theory and Psychophysics*. John Wiley and Sons, New York NY, 1966.

[Has09] Ahmed Hassan. Predicting faults using the complexity of code changes. In *Proc. Int'l Conf. on Softw. Eng.*, pages 78–88, 2009.

[Hed08] Larry Hedges. What are effect sizes and why do we need them? *Child Development Perspectives*, 2(3):167–171, 2008.

[HH03] Mark Hall and Geoffrey Holmes. Benchmarking attribute selection techniques for discrete class data mining. *IEEE Trans. on Knowl. and Data Eng.*, 15(6):1437–1447, November 2003.

[HH05] Ahmed Hassan and Richard Holt. The top ten list: Dynamic fault prediction. In *Proc. Int'l Conf. on Softw. Eng.*, pages 263–272, 2005.

[HmL05] Shih-Kun Huang and Kang min Liu. Mining version histories to verify the learning process of legitimate peripheral participants. In *Proc. Int'l Workshop on Mining Softw. Repositories*, pages 1–5, 2005.

[HW07] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Proc. Int'l Conf. on Automated Softw. Eng.*, pages 34–43, 2007.

[JV09] Natalia Juristo and Sira Vegas. Using differences among replications of software engineering experiments to gain knowledge. In *Proc. Int'l Symposium on Empirical Softw. Eng. and Measurement*, pages 356–366, 2009.

[KAB+96] T. M. Khoshgoftaar, E. B. Allen, L. A. Bullard, R. Halstead, and G. P. Trio. A tree-based classification model for analysis of a military software system. In *Proc. High-Assurance Systems Engineering Workshop*, pages 244–251, 1996.

[KAG+96] Taghi Khoshgoftaar, Edward Allen, Nishith Goel, Amit Nandi, and J McMullan. Detection of software modules with high debug code churn in a very large legacy system. In *Proc. Int'l Symposium on Softw. Reliability Eng.*, page 364, 1996.

[Kas80] G. V. Kass. An exploratory technique for investigating large quantities of categorical data. *Journal of Applied Statistics*, 29(2):119–127, 1980.

[KCM07] Huzefa Kagdi, Micheal Collard, and Jonathan Maletic. A survey and taxonomy of approaches for mining software repositories in

the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, March 2007.

[KDHS07] Vigdis Kampenes, Tore Dyba, Jo Hannay, and Dag Sjoberg. A systematic review of effect size in software engineering experiments. *Information and Software Technology*, 49(11-12):1073 – 1086, 2007.

[Ker79] Frederick Kerlinger. *Behavioral Research: A Conceptual Approach*. Harcourt School, 1979.

[KFN99] Cem Kaner, Jack Falk, and Hung Nguyen. *Testing Computer Software*. Wiley, 2nd edition, April 1999.

[KHL+98] Harvey Keselman, Carl Huberty, Lisa Lix, Stephen Olejnik, Robert Cribbie, Barbara Donahue, Rhonda Kowalchuk, Laureen Lowman, Martha Petoskey, Joanne Keselman, and Joel Levin. Statistical practices of educational researchers: An analysis of their anova, manova, and ancova analyses. *Review of Educational Research*, 68(3), 1998.

[KMC06] Andrew Ko, Brad Myers, and Duen Horng Chau. A linguistic analysis of how people describe software problems. In *Proc. Visual Languages and Human-Centric Computing*, pages 127–134, 2006.

[KMM+10] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Kenichi Matsumoto, Bram Adams, and Ahmed Hassan. Revisiting common bug prediction findings using effort-aware models. In *Proc. Int'l Conf. on Softw. Maint.*, pages 1–10, 2010.

[Koh95] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proc. of the Int'l Joint Conf. on Artificial Intelligence*, pages 1137–1143. Morgan Kaufmann, 1995.

[KPB06] Patrick Knab, Martin Pinzger, and Abraham Bernstein. Predicting defect densities in source code files with decision tree learners. In *Proc. Int'l Workshop on Mining Software Repositories*, pages 119–125, 2006.

[KR11] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *Proc. Int'l Conf. on Softw. Eng.*, pages 351–360, 2011.

[KS94] Taghi Khoshgoftaar and Robert Szabo. Improving code churn predictions during the system test and maintenance phases. In *Proc. Int'l Conf. on Softw. Maintenance*, ICSM '94, pages 58–67. IEEE Press, 1994.

[KW06] Sunghun Kim and E. James Whitehead, Jr. How long did it take to fix bugs? In *Proc. Int'l Workshop on Mining Softw. Repos.*, pages 173–174, 2006.

[KWZ08] Sunghun Kim, James Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34(2):181–196, March 2008.

[KZPW06] Sunghun Kim, Thomas Zimmermann, Kai Pan, and James Whitehead. Automatic identification of bug-introducing changes. In *Proc. Int'l Conf. on Automated Softw. Eng.*, pages 81–90, 2006.

[KZWG11] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *Proc. Int'l Conf. on Softw. Eng.*, pages 481–490, 2011.

[KZWZ07] Sunghun Kim, Thomas Zimmermann, James Whitehead, and Andreas Zeller. Predicting faults from cached history. In *Proc. Int'l Conf. on Softw. Eng.*, pages 489–498, 2007.

[Lar06] Daniel Larose. *Data Mining Methods and Models*. John Wiley and Sons, January 2006.

[LBN09] Rajesh Vasaand Markus Lumpe, Philip Branch, and Oscar Nierstrasz. Comparative analysis of evolving software systems using the gini coefficient. In *Proc. Int'l Conf. on Softw. Maintenance*, pages 179 –188, 2009.

[LBSP08] Stefan Lessmann, Bart Baesens, Christophe Mues Swantje, and Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. on Softw. Eng.*, 34:485–496, July 2008.

[LH93] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.

[LNH+11] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. Micro interaction metrics for defect prediction. In *Proc. Joint European Softw. Eng. Conf. and ACM SIGSOFT Symposium on the Foundations of Softw. Eng.*, pages 311–321, 2011.

[Lor05] Max Otto Lorenz. Methods of measuring the concentration of wealth. *Publications of the American Statistical Association*, 9(70):209–219, June 1905.

[McK84] James McKee. Maintenance as a function of design. In *Proc. National computer Conference and Exposition*, pages 187–193, 1984.

[MDDG07] Tim Menzies, Alex Dekhtyar, Justin Distefano, and Jeremy Greenwald. Problems with precision: A response to "comments on 'data mining static code attributes to learn defect predictors'". *IEEE Trans. Softw. Eng.*, 33:637–640, September 2007.

[MFH02] Audris Mockus, Roy Fielding, and James Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, July 2002.

[MGF07] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. on Softw. Eng.*, 33:2–13, January 2007.

[Mil00] James Miller. Applying meta-analytical procedures to software engineering experiments. *Journal of Systems and Softw.*, 54(1):29–39, 2000.

[Mil05] James Miller. Replicating software engineering experiments: a poisoned chalice or the holy grail. *Information and Software Technology*, 47(4):233 – 244, 2005.

[MMT+10] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Softw. Eng.*, 17(4):375–407, 2010.

[MPS08] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proc. Int'l Conf. on Softw. Eng.*, pages 181–190, 2008.

[MW00] Audris Mockus and David Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5:169–180, 2000.

[MWK+06] Ingo Mierswa, Micheal Wurst, Ralf Klinkenberg, Martin Scholz, and Timm Euler. Yale: Rapid prototyping for complex data mining tasks. In *Proc. ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, pages 935–940. ACM, 2006.

[NAH10] Thanh Nguyen, Bram Adams, and Ahmed Hassan. Studying the impact of dependency network measures on software quality. In *Int'l Conf. on Softw. Maintenance*, pages 1 –10, 2010.

[NB05] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proc. Int'l Conf. on Softw. Eng.*, pages 284–292, 2005.

[NBZ06] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proc. Int'l Conf. on Softw. Eng.*, pages 452–461, 2006.

[NMB04] Tien Nguyen, Ethan Munson, and John Boyland. The molhado hypertext versioning system. In *Proc. ACM Conf. on Hypertext and Hypermedia*, pages 185–194, 2004.

[NMB08] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality: an empirical case study. In *Proc. Int'l Conf. on Softw. Eng.*, pages 521–530, 2008.

[Nor10] Marija Norusis. *SPSS 18 Advanced Statistical Procedures Companion*. Pearson, March 2010.

[NZZ+10] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change bursts as defect predictors. In *Proc. Int'l Symposium on Softw. Reliability Eng.*, 2010.

[OA96] Niclas Ohlsson and Hans Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Trans. on Softw. Eng.*, 22(12):886–894, Dec. 1996.

[OOOM05] Masao Ohira, Naoki Ohsugi, Tetsuya Ohoka, and Kenichi Matsumoto. Accelerating cross-project knowledge collaboration using collaborative filtering and social networks. In *Proc. Int'l Workshop on Mining Softw. Repositories*, pages 1–5, 2005.

[OWB05] Thomas Ostrand, Elaine Weyuker, and Robert Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4):340–355, 2005.

[PA06] Shari Pfleeger and Joanne Atlee. *Software Engineering - Theory and Practice*. Pearson Education, 3 edition, 2006.

[Pan07] Lucas Panjer. Predicting eclipse bug lifetimes. In *Proc. Int'l Workshop on Mining Softw. Repos.*, pages 29–32, 2007.

[Par94] David Parnas. Software aging. In *Proc. Int'l Conf. on Softw. Eng.*, pages 279–287, 1994.

[PC86] David Parnas and Paul Clements. A rational design process: How and why to fake it. *IEEE Trans. Softw. Eng.*, 12(2):251–257, February 1986.

[PFD11] Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. Ecological inference in empirical software engineering. In *Proc. Int'l Conf. on Automated Softw. Eng.*, pages 362–371, 2011.

[Pig96] Thomas Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. Wiley, 1 edition, October 1996.

[PNM08] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can developer-module networks predict failures? In *Proc. ACM SIGSOFT Int'l Symposium on the Foundations of Softw. Eng.*, pages 2–12, 2008.

[PP05] Ranjith Purushothaman and Dewayne Perry. Toward understanding the rhetoric of small source code changes. *IEEE Trans. Softw. Eng.*, 31(6):511–526, June 2005.

[RAN07] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *Proc. Int'l Conf. on Softw. Eng.*, pages 499–510, 2007.

[RD11] Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proc. Int'l Conf. on Softw. Eng.*, pages 491–500, 2011.

[RL08] Romain Robbes and Michele Lanza. Spyware: a change-aware development toolset. In *Proc. Int'l Conf. on Softw. Eng.*, pages 847–850, 2008.

[RN78] Barry De Roze and Thomas Nyman. The software life cycle—a management and technological challenge in the department of defense. *IEEE Trans. on Softw. Eng.*, SE-4(4):309–318, July 1978.

[Rom87] Dieter Rombach. A controlled expeniment on the impact of software structure on maintainability. *IEEE Trans. on Softw. Eng.*, 13:344–354, 1987.

[RPL08] Romain Robbes, Damien Pollet, and Michele Lanza. Logical coupling based on fine-grained change information. In *Proc. Working Conf. on Reverse Eng.*, pages 42–46, 2008.

[Sch93] William Schafer. Interpreting statistical significance and non-significance. *The Journal of Experimental Education*, 61(4):383–387, 1993.

[SJI+10] Emad Shihab, Ming Jiang, Walid Ibrahim, Bram Adams, and Ahmed Hassan. Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In *Proc. Int'l Symposium on Empirical Softw. Eng. and Measurement*, pages 1–10, 2010.

[SK03] Ramanath Subramanyam and M.S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310, 2003.

[SLM03] Sayyad Shirabad, Timothy Lethbridge, and Stan Matwin. Mining the maintenance history of a legacy software system. In *Proc. Int'l Conf. on Softw. Maintenance*, pages 95–104, 2003.

[SMK+11] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed Hassan. High-impact defects: a study of breakage and surprise defects. In *Proc. Joint European Softw. Eng. Conf. and ACM SIGSOFT Symposium on the Foundations of Softw. Eng.*, pages 300–310, 2011.

[Som00] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6 edition, 2000.

[SP88] Richard Selby and Adam Porter. Learning from examples: generation and evaluation of decision trees for software resource analysis. *IEEE Trans. on Softw. Eng.*, 14(12):1743–1757, 1988.

[SvdB10] Alexander Serebrenik and Mark van den Brand. Theil index for aggregation of software metrics values. In *Proc. Int'l Conf. on Softw. Maintenance*, pages 1–9, 2010.

[Swa76] Burton Swanson. The dimensions of maintenance. In *Proc. Int'l Conf. on Softw. Eng.*, pages 492–497, 1976.

[SZZ05] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proc. Int'l Workshop on Mining Software Repositories*, pages 1–5, 2005.

[SZZ06] Adrian Schroeter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proc. Int'l Symposium on Empirical Softw. Eng.*, pages 18–27, 2006.

[Tas02] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002.

[TCS05] Nikolaos Tsantalis, Alexander Chatzigeorgiou, and George Stephanides. Predicting the probability of change in object-oriented systems. *IEEE Trans. Softw. Eng.*, 31(7):601–614, 2005.

[TMBS09] Burak Turhan, Tim Menzies, Ayşe Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Softw. Eng.*, 14(5):540–578, October 2009.

[Vuj08] Aleks Vujanic. Most u.s. companies say business analytics still future goal, not present reality. In *Accenture Newsroom*. Accenture Ltd., Dec. 2008.

[WF94] Stanley Wasserman and Katherine Faust. *Social Network Analysis: Methods and Applications*. Camb. Univ. Press, 1 edition, 1994.

[WF05] Ian Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Data Management Systems. Morgan Kaufmann, second edition, June 2005.

[Win08] Rory Winston. The gini coefficient as a measure of software project risk. http://www.theresearchkitchen.com/blog/archives/219, September 2008.

[WOB08] Elaine Weyuker, Thomas Ostrand, and Robert Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Softw. Eng.*, 13(5):539–559, October 2008.

[WPZZ07] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Proc. Int'l Workshop on Mining Softw. Repos.*, pages 1–10, 2007.

[WT99] Leland Wilkinson and Task Force on Statistical Inference. Statistical methods in psychology journals: Guidelines and explanations. *American Psychologist*, 54(8):594–604, 1999.

[WZKC11] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. Relink: Recovering links between bugs and changes. In *Proc. Joint European Softw. Eng. Conf. and ACM SIGSOFT Symposium on the Foundations of Softw. Eng.*, pages 15–25, 2011.

[WZX+08] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proc. of the Int'l Conf. on Softw. Eng.*, pages 461–470, 2008.

[YMNCC04] Annie Ying, Gail Murphy, Raymond Ng, and Mark Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, 2004.

[Zha09] Hongyu Zhang. An investigation of the relationships between lines of code and defects. In *Proc. Int'l Conf. on Softw. Maint.*, pages 274–283, 2009.

[ZL07] Yuming Zhou and Hareton Leung. Predicting object-oriented software maintainability using multivariate adaptive regression splines. *Journal of Systems and Software*, 80(8):1349–1361, 2007.

[ZN08] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proc. Int'l Conf. on Softw. Eng.*, pages 531–540, 2008.

[ZNG+09] Thomas Zimmermann, Nachiappan Nagappan, Harald C. Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proc. Joint European Softw. Eng. Conf. and ACM SIGSOFT Symposium on the Foundations of Softw. Eng.*, pages 91–100, 2009.

[ZPZ07] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proc. Int'l Workshop on Predictor Models in Softw. Eng.*, pages 9–15, 2007.

[ZWDZ04] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proc. Int'l Conf. on Softw. Eng.*, pages 563–572, 2004.

[ZZ07] Hongyu Zhang and Xiuzhen Zhang. Comments on "data mining static code attributes to learn defect predictors". *IEEE Trans. Softw. Eng.*, 33:635–637, September 2007.

Curriculum Vitae

Personal	**Information**
Name	Emanuel Giger
Nationality	Swiss
Date of birth	October 21, 1980
Place of origin	Zurich, ZH, Switzerland

Education

2007–2012	*Doctor of Science* Department of Informatics, University of Zurich, Switzerland Subject of dissertation: "Fine-Grained Code Changes and Bugs: Improving Bug Prediction" Advisors: Prof. Dr. Harald C. Gall, University of Zurich Prof. Dr. Andreas Zeller, Saarland University
2001–2006	*Dipl. Inform.* Department of Informatics, University of Zurich, Switzerland Subject of master thesis: "Evolving Code Clones" Advisor: Prof. Dr. Harald C. Gall, University of Zurich
1994–2001	*Maturität Typus B* Literargymnasium Rämibühl Zurich, Switzerland
1993–1994	Sekundarschule Zurich, Switzerland
1987–1993	Primarschule Zurich, Switzerland